Einführung in Forth-83

Teil 1

Dr. Hartmut Pfüller (Leiter), Dr. Wolfgang Drewelow, Dr. Bernhard Lampe, Ralf Neuthe, Egmont Woitzel
Wilhelm-Pieck-Universität Rostock, Sektion Technische Elektronik

Gliederung

1. Einführung und Kernwortschatz

Die dialogorientierte Programmiersprache Forth wurde in den sechziger Jahren von Charles H. Moore in den USA entwickelt und ab 1971 zunächst für die Echtzeitsteuerung von Radioteleskopen eingesetzt. Entwicklungsziele waren maximale Handlichkeit der Sprache zwecks hoher Produktivität des Programmierers und größte Einfachheit des Übersetzerkonzepts. Herausgekommen ist eine organische und kompakte Einheit von Sprache und Übersetzer. Forth ist erweiterbar, das heißt, der Quelltext kann für sich selbst die eigenen Ausdrucksmittel ändern und neue Ausdrucksmittel höheren Niveaus erzeugen. Damit ist problemnahe Programmierung für nahezu jede Anwendung in einheitlicher Softwareumgebung und *in einem Zuge* möglich, also ohne Mehrpaßübersetzung. In Richtung niederen Niveaus sind im Quelltext die Maschinenbefehle des Wirtsprozessors verwendbar. Das ist nützlich für direkte Gerätesteuerungen und Zugriffe auf Treiber.

Forthsysteme sind je nach Ausstattung etwa drei KByte bis über 16 KByte groß und für praktisch alle Rechner verfügbar. Forth wurde inzwischen in mehreren Etappen standardisiert; die vorliegende Beschreibung folgt dem Standard Forth-83. Eventuelle Unklarheiten beim Nachvollziehen der Beispiele sollten Veranlassung sein, die Verträglichkeit des benutzten Systems mit diesem Standard zu überprüfen.

1.1. Die Softwarekonzeption von Forth

1.1.1. Die Architektur von Forth

Das Forthsystem ist ein Rechenprogramm, das gleichzeitig als Betriebssystem, als Compiler und als Kommandoprozessor (*Textinterpreter*) arbeitet. Bild 1.1 soll das an einem Schichtenmodell anschaulich machen. Die niedrigste Schicht enthält alle Programmmoduln, die aus Maschinencode gebildet sind. Dieses Prinzip, die Gegebenheiten einer konkreten Hardware mittels eines zugeordneten Pakets von Maschinenprogrammen *abzufangen*, ist mit dem BIOS-Teil des Betriebssystems CP/M vergleichbar. Auch die nächsthöheren Schichten sind in Moduln gegliedert, allerdings bestehen diese nicht aus Maschinencode, sondern aus sogenanntem Forthcode. Das sind Listen von Adressen, wobei jede Adresse als Pointer zu einem bestimmten Modul aus Maschinencode oder aus Forthcode verweist. Die Betriebssystemschicht und die darüberliegenden sind dadurch maschinenunabhängig, also portabel. Alle Schichten können jederzeit auch im Dialog erweitert werden.

1.1.2. Das Wortkonzept

Das gesamte Forthsystem besteht aus einer größeren Anzahl (je nach Systemgröße z. B. etwa 70...300) von relativ kleinen Moduln (teils aus Maschinencode, teils aus Forthcode). Alle diese Grundfunktionen sind im Hauptspeicher lexikonförmig geordnet aufbewahrt. Wegen der tatsächlichen Ähnlichkeit mit einem Lexikon heißt dieser Programmteil *Wörterbuch*, und jeder der einzelnen Moduln wird als *Wort* bezeichnet. Die Worte im Wörterbuch dienen als Kommunikationsmittel zwischen Mensch und Rechner und müssen demzufolge von beiden *verstanden* werden. Zu diesem Zweck ist jeder Eintrag zweigeteilt und so aufgebaut, daß der Name des Wortes für den Menschen als Text lesbar und der zugehörige Codeteil für den Rechner ausführbar ist. Vier Beispiele für Namen sind:

; */MOD 2! VOCABULARY

Die Namensbildung ist sehr freizügig; Sonderzeichen sind an beliebiger Stelle erlaubt oder können auch einzeln als Name gelten.

1.1.3. Das Stapelkonzept

Außer Namen können im Eingabetext von Tastatur oder Massenspeicher natürlich auch Zahlen erscheinen. Diese Zahlen werden eine nach der anderen in das *interne Format* konvertiert, so zum *Parameterstapel* (Parameterstack) geschafft und dort abgelegt. Zur internen Wertdarstellung dienen durchgängig 16-Bit-Dualzahlen; für negative Zahlen wird die Zweierkomplementdarstellung verwendet. Auf dem Parameterstapel bleiben die Werte dann so lange liegen, bis sie von irgendeinem Wort wieder abgeholt (*verbraucht*) werden. Damit ist gleichzeitig angedeutet, wie Forthworte mit lokalen Eingangsparametern versorgt werden: Die Parameter müssen irgendwann vor Aktivierung des Wortes auf dem Parameterstapel hinterlegt worden sein. Falls mit mehreren Werten hantiert wird, gilt das LIFO-Prinzip (englisch: last in – first out). Entsprechend wird mit lokalen Ausgangsparametern verfahren: Falls ein Wort lokale Ergebniswerte liefert, bleiben diese nach Ausführung des Wortes auf dem Stapel liegen und sind so zum Beispiel wieder als Eingangsparameter für nachfolgende Worte verfügbar. Da auf diese Weise bei Aufrufen die aus anderen Programmiersprachen bekannten Listen von aktuellen und formalen Parametern entfallen, spricht man bei Forth auch von *impliziter Parameterübergabe*. Für Programmdokumentationen oder Quelltextkommentare kann es erforderlich sein, genauere Angaben über Anzahl und Art der lokalen Ein- und Ausgangsparameter zu machen. Dafür hat sich in Forth eine bestimmte Art der Kommentierung eingebürgert: In runden Kommentarklammern wird notiert, wie der Stapel vor und nach Ausführung des kommentierten Wortes belegt ist. Die Notation

(n1 n2 n3 ===> n4 n5)

bedeutet in diesem Sinne, daß das kommentierte Forthwort drei Eingangsparameter vom Stapel holt – mit n3 an der Stapelspitze – und zwei neue Werte – mit n5 an der Stapelspitze – als Ausgangsparameter hinterläßt. Der Pfeil symbolisiert die Ausführung des Wortes.

1.1.4. Das Erweiterungskonzept

Die Vorgehensweise beim Programmieren in Forth soll an einem Steuerprogramm für einen hypothetischen x-y-Schreiber erläutert werden. Dieser Plotter sei so einfach, daß seine Hardware nur drei Funktionen kennt, und zwar

a) *Stift anheben*
b) *Stift absenken*
c) *geradenwegs die Absolutposition (x, y) anfahren.*

Wie diese Funktionen von der Rechnerperipherie aus zu aktivieren sind, soll bekannt sein. Programmieren in Forth heißt nun, das Forthsystem zu erweitern, indem der Programmierer zu den im Wörterbuch schon existierenden Worten neue Worte *hinzudefiniert*, nämlich solche, die Schritt für Schritt die Programmieraufgabe lösen. Dabei können alle schon im Wörterbuch existierenden Worte ausgenutzt werden. Zum Definieren neuer Worte gibt es (natürlich *auch* im Wörterbuch!) *Definitionsworte*. Ein neues Wort für die Maschinencodeschicht wird zum Beispiel durch Voranstellen des Definitionswortes **CODE** vor einen selbstzuwählenden Namen erzeugt. Mit dem Wort **END-CODE** wird ein solches Maschinenprogramm dann wieder beendet. Der Forthprogrammierer geht nun üblicherweise so vor, daß er entsprechend den Gerätedaten das Assemblerprogramm konzipiert und dann eintippt:

```
CODE HEBEN  . . . . . . . . . .  END-CODE
CODE SENKEN . . . . . . . . . .  END-CODE
```

Da die konkreten Assemblerbefehle hier nicht im Mittelpunkt stehen, wurden sie nur



Kurs

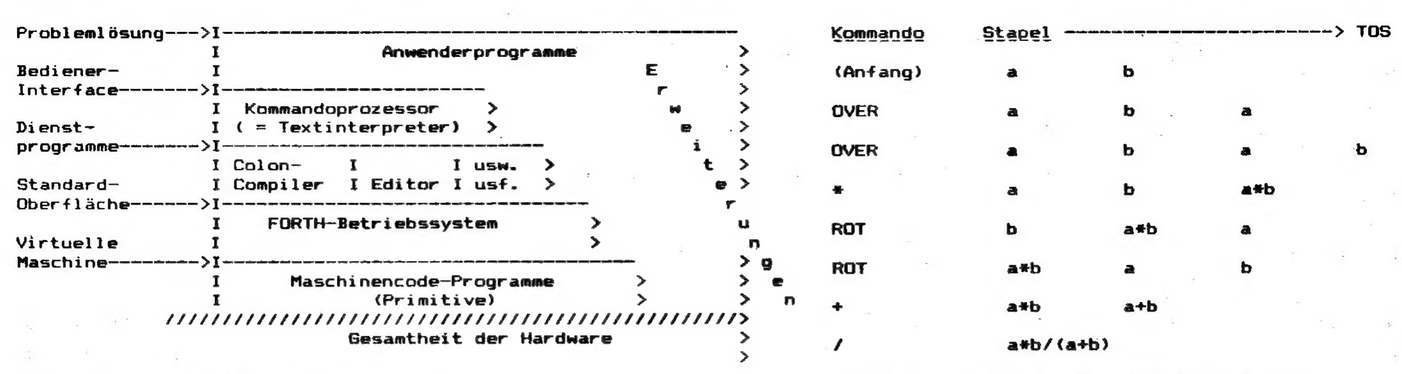


Bild 1.1 Schichtenmodell eines Forthsystems

Kommando	Stapel	---------------------->	TOS	
(Anfang)	a	b		
OVER	a	b	a	
OVER	a	b	a	b
*	a	b	a*b	
ROT	b	a*b	a	
ROT	a*b	a	b	
+	a*b	a+b		
/	a*b/(a+b)			

Bild 1.5 Stapelbelegung während der Formelberechnung

```
2233  -77  444  (cr) ok            (wird auf Stapel gelegt)
 .  (cr) 444 ok                    (letzter Wert entnommen)
 .  .  (cr) -77 2233 ok            (nächste Werte entnommen)
```

Bild 1.2 Ausgabe von Zahlen mittels Punktbefehl

```
1  65535  -2  (cr) ok
U.  U.  U.  (cr) 65534 65535 1 ok
```

Bild 1.3 Vorzeichenlose Ausgabe von Zahlen

```
0  1  D.  (cr) 65536 ok
1  0  D.  (cr) 1 ok
```

Bild 1.4 Ausgabe von doppeltgenauen Zahlen

durch Punkte angedeutet. Nachdem nun die neuen Worte **HEBEN** und **SENKEN** definiert sind, kann der Programmierer durch deren Aufruf bei angeschlossenem Plotter kontrollieren, inwieweit die von ihm beabsichtigte Funktion korrekt ausgeführt wird. Die *Ausführung* von Worten erreicht man einfach durch Eintippen ihres Namens und Bestätigung der Eingabe mit der <Enter>-Taste. Diese *normale* Betriebsart von Forth heißt deshalb auch *Ausführungsmodus*. Wenn die neuen Worte augenscheinlich nicht korrekt arbeiten, müssen sie berichtigt neu eingegeben werden, bis der gewünschte Erfolg erreicht ist. Danach werde in ähnlicher Weise die dritte Funktion als Maschinenprogramm codiert:

CODE ANFAHREN . . . . END-CODE

Der Programmierer hat **ANFAHREN** als sinnfälligen Namen für diese Funktion gewählt. Hier ist nun zu beachten, daß dieses Wort die Parameter x und y für die Zielkoordinaten benötigt. In Quellprogrammen gehört es zu gutem Forthstil, mit dem neudefinierten Namen entsprechende Hinweise als Kommentar in runden Klammern zu notieren.

```
CODE ANFAHREN    ( x y ===>)
      . . . . . . . . . . .  END-CODE
```

Durch die Ausführung des Wortes **ANFAHREN** werden also zwei Werte vom Stapel entfernt (verbraucht). Auch die Funktion dieses Wortes kann nun im Dialog getestet werden, z. B. durch Eintippen von:

50 70 ANFAHREN

Wenn das Maschinenprogramm **ANFAHREN** korrekt ist, muß damit die Position (x = 50, y = 70) angefahren werden. Natürlich können Worte auch im Verbund arbeiten, zum Beispiel so:

```
HEBEN  0  0  ANFAHREN
SENKEN  100  50  ANFAHREN  HEBEN
```

Nach Eingabe dieser Zeilen muß der Plotter eine Strecke vom Punkt (x = 0, y = 0) zum Punkt (x = 100, y = 50) zeichnen. Neben dem Definitionswort **CODE** für das Eintragen von Maschinencodeworten ins Wörterbuch gibt es weitere Definitionsworte. Das vielleicht wichtigste von ihnen ist der *Doppelpunkt*. Er dient zur Erweiterung der höheren Schichten um Moduln in Forthcode. Für den Plotter kann zum Beispiel ein Wort definiert werden, dessen Funktion es ist, zwei Punkte durch eine Linie zu verbinden:

```
:  VERBINDEN   ( x2 y2 x1 y1 ===>)
   ANFAHREN SENKEN ANFAHREN HEBEN ;
```

Hier werden die Worte **ANFAHREN**, **SENKEN** und **HEBEN** nach Eingabe *nicht* ausgeführt, sondern als Programm in Forthcode unter dem neuen *Stichwort* **VERBINDEN** ins Wörterbuch kompiliert. Das wird dadurch erreicht, daß der Doppelpunkt generell nach seiner Aktivierung die Betriebsart vom Ausführungsmodus in den sogenannten *Kompilationsmodus* umschaltet. Das abschließende Semikolon (*auch* ein Forthwort!) schaltet dann wieder vom Kompilationsmodus in den Ausführungsmodus zurück. Die Gesamtkonstruktion einschließlich Doppelpunkt und Semikolon wird Doppelpunktdefinition genannt. Nach der Definition kann natürlich auch das Wort **VERBINDEN** im Dialog getestet werden, zum Beispiel so:

80 100 0 20 VERBINDEN

Die Ausführung des Wortes **VERBINDEN** bedeutet nun, daß genau diejenigen Befehle ausgeführt werden, die in der Doppelpunktdefinition notiert sind, also: **ANFAHREN SENKEN ANFAHREN HEBEN**. Man mache sich klar, daß die Richtung vom Punkt (x = 0, y = 20) zum Punkt (x = 80, y = 100) führt: Die Koordinaten des Punktes (x = 0, y = 20) liegen obenauf und werden deshalb als erste vorgefunden und angefahren. In ähnlicher Weise können im Dialogbetrieb nach und nach weitere Worte definiert, schrittweise zu komplexeren Funktionen zusammengefaßt und getestet werden. Zum Beispiel könnte man die Worte definieren, die lediglich die Parameter von markanten Punkten des Achsenkreuzes liefern:

```
: NULLPUNKT  ( ===> x0 y0)   0   0 ;
: XMAX  ( ===> xmax y0)  100   0 ;
: YMAX  ( ===> x0 ymax)   0  100 ;
```

Das erlaubt dann sinnfällige Wortsequenzen wie

. . . NULLPUNKT ANFAHREN . . .

Weiter könnte damit nun beispielsweise ein Wort zum Zeichnen der positiven Achsen definiert werden:

```
: ACHSEN   ( ===>)
  NULLPUNKT  YMAX  VERBINDEN
  XMAX  NULLPUNKT  VERBINDEN ;
```

Wenn Programme ausprobiert werden sollen, ohne daß das anzusprechende Gerät verfügbar ist, kann man sich dadurch helfen, daß die Grundfunktionen durch Doppelpunktworte definiert werden. Dabei muß das Stapelverhalten korrekt nachgebildet werden. Anstatt die eigentliche Funktion auszuführen, können zum Beispiel Zeichenkettentexte auf dem Bildschirm angezeigt werden:

```
: HEBEN      ." heben " ;
: SENKEN     ." senken " ;
: ANFAHREN   ( x y ===>)
   .  .      ." anfahren " ;
```

Nach dem Eintippen dieser Worte könnten die obigen Beispiele zur Plottersteuerung nachvollzogen werden.

1.2. Das Bedienerinterface

1.2.1. Integerzahlen

Der Textinterpreter von Forth versucht für jeden (in Leerzeichen eingegrenzten) Eintrag zunächst, die Zeichenfolge als Name eines Forthwortes im Wörterbuch aufzufinden. Falls das mißlingt, wird als nächstes geprüft, ob die eingebene Zeichenfolge als Zahl verstanden werden kann, das heißt, ob sie ausschließlich aus gültigen Ziffernzeichen (allenfalls mit führendem Minuszeichen) besteht. Wenn diese Prüfung erfolgreich ist, wird die Ziffernzeichenfolge in die rechnerinterne Zahlenwertdarstellung umgewandelt und so auf dem Parameterstapel abgelegt.

1.2.2. Worte

Jedes eingegebene Wort, ob es nun eine (ein- oder mehrstellige) Zahl oder ein im Wörterbuch enthaltenes Forthwort repräsentiert, muß vom nachfolgenden Wort im Programmtext durch mindestens ein Leerzeichen getrennt sein. Nachfolgend wird die Benutzung von Zahlen und Forthworten am Beispiel einiger Ausgabebefehle demonstriert.

Wortname: .
Stapeleffekt: (n ===>)
Funktion: Das ASCII-Zeichen *Punkt* ist in Forth der einfache Ausgabebefehl; er holt (d. h. entfernt) den obersten 16-Bit-Wert vom

16-Bit-Wert vom Parameterstapel, faßt ihn als vorzeichenbehaftete Zahl in Zweierkomplementdarstellung (−32768 ... 32767) auf, wandelt ihn in die externe Ziffernzeichendarstellung und sendet diese ASCII-Zeichenkette als externe Darstellung der Dualzahl zum Ausgabegerät (Bild 1.2).
Zur Unterscheidung von den Bedienereingaben ist die Rechnerreaktion in diesen Dialogmitschnitten jeweils durch Unterstreichung gekennzeichnet. Das zusätzlich eingefügte (cr) soll bedeuten, daß vor dieser Stelle die Entertaste betätigt wurde.
Wortname: **U.**
Stapeleffekt: (u ===>)
Funktion: Ausgabebefehl wie Punkt; faßt den Wert als vorzeichenlose Dualzahl (0 ... 65536) auf (Bild 1.3).
Wortname: **D.**
Stapeleffekt: (*n n* ===>) oder identisch: (d ===>)
Funktion: Ausgabebefehl; holt die oberen beiden (16-Bit-)Werte vom Stapel und faßt dieselben gemeinsam als 32-Bit-Ganzzahl in vorzeichenbehafteter Zweierkomplementdarstellung (−2147483648 bis 2147483647) auf (Bild 1.4).
Eine Kurzbeschreibung aller aufgeführten Befehle ist in Tafel 1.2 zu finden. Da diese in vielen Fällen ausreichend ist, wird es im weiteren Verlauf seltener erforderlich sein, Worte so ausführlich wie eben vorgeführt zu beschreiben. In allen drei obigen Beschreibungen wurde absichtlich dazugesagt, wie der Ausgabebefehl den Stapelwert *auffaßt*. Diese *Auffassungsfrage* gilt nicht nur für Ausgabebefehle, sondern generell: Auf dem Stapel werden physisch nur 16-Bit-Werte verwaltet. Der Programmierer allein entscheidet bei neuen Definitionen durch die von ihm organisierte Weiterverwendung darüber, ob sein Forthwort einen Wert als vorzeichenbehaftet, als ASCII-Zeichen mit nur sieben gültigen Bits oder mit weiteren Parametern gemeinsam als mehrfachgenauen Wert benutzt oder auf völlig andere Weise. Bei der Verwendung von bereits existierenden Worten muß er sich natürlich darüber informieren, welche Bedeutung diese jeweils den Parametern beilegen.

1.3. Der Parameterstapel

Da die Werte auf dem Parameterstapel für sehr verschiedene Anwendungsfälle als Eingangsparameter verwendet werden können, ist es manchmal zweckmäßig, vorhandene Werte zu kopieren oder umzusortieren. Dafür sind die in den nachfolgenden Unterabschnitten aufgeführten Worte nützlich.

1.3.1. Vervielfältigung

Die sechs Worte
DUP OVER PICK ?DUP 2DUP 2OVER
stellen alle in irgendeiner Weise Kopien von bestimmten Werten irgendwo auf dem Stapel her, die dann als Ergebnisparameter auf das obere Stapelende (englisch: Top of stack, Abk.: TOS) abgelegt werden. Die Stapeldiagramme und Kurzbeschreibungen sind in Tafel 1.2 enthalten. Mit **PICK** kann ein beliebiger Wert zum TOS kopiert werden, dessen Position man allerdings explizit angeben muß. **?DUP** dupliziert den Wert im TOS genau dann, wenn er nicht gleich Null ist. Das ist besonders in Verbindung mit Entscheidungsstrukturen nützlich.

1.3.2. Ordnungsbefehle

Jedes der fünf Worte
ROT SWAP ROLL 2ROT 2SWAP
leistet in irgendeiner Weise eine Umsortierung von Werten auf dem Parameterstapel; der Füllstand des Stapels wird dabei nicht geändert.

1.3.3. Entfernen von Werten

Die Worte **DROP** und **2DROP** entfernen vom TOS einen 16-Bit-Wert bzw. einen 32-Bit-Wert. Das wird zum Beispiel dann benötigt, wenn erst nach Entscheidungsfolgen klar wird, welcher von mehreren Parametern weiter benötigt wird und welcher nicht. Überflüssige Werte können dann durch Umsortierung nach oben gebracht und mit diesen Befehlen *vernichtet* werden.

1.3.4. Stapeltiefe

Das Wort **DEPTH** (deutsch: Tiefe) liefert als Ergebnisparameter eine Zahl, die angibt, mit wieviel 16-Bit-Werten der Parameterstapel gefüllt ist.

1.4. Arithmetische Operationen

1.4.1. Umgekehrte polnische Notation

Im Plotterbeispiel wurde eine Befehlsfolge **NULLPUNKT ANFAHREN** benutzt. Dabei folgt auf den (zusammengesetzten) *Parameter* **NULLPUNKT** die *Operation* **ANFAHREN**. Nach diesem Prinzip kann auch mit Zahlen operiert werden. Zum Beispiel gibt es in Forth eine Operation für das Addieren; das entsprechende Forthwort hat den Namen + erhalten. Angenommen, auf dem Stapel liege ein Wert **5**, dann führt die Befehlsfolge **3 +** dazu, daß die **5** um den Wert **3** erhöht, also zu einer **8** wird. Wesentlich an dieser Betrachtung ist, daß in Forth generell die Operation *hinter* dem (oder den) Operanden steht (sogenannte Postfixnotation; andere übliche Bezeichnungen dafür sind UPN = umgekehrte polnische Notation oder englisch: RPN = reverse Polish Notation). In Verbindung mit der LIFO-Verwaltung des Parameterstapels wird sich das als sehr praktisch erweisen. Trotzdem ist es etwas unüblich, denn die in der Schule gelehrte Notation setzt den Operator *zwischen* die Operanden (sogenannte Infixnotation). Es läßt sich zeigen, daß daneben auch noch eine Präfixnotation möglich ist (Operator *vor* den Operanden; z. B. von der Programmiersprache LISP bevorzugt) und daß alle diese Notationen ineinander überführbar sind. Ungewohnt ist die Postfixnotation am Anfang vielleicht am ehesten bei größeren Formeln.

1.4.2. Die Grundrechenarten

Tafel 1.2 zeigt die Funktion der sechs Worte
+ − * / D+ D−
Die ersten vier erwarten jeweils zwei Parameter auf dem Stapel, entfernen diese und hinterlassen im TOS das Ergebnis der Operation. Der Programmierer muß wissen, daß Bereichsüberschreitungen zum Beispiel bei Addition oder Multiplikation nicht reklamiert werden. Weiter ist wichtig zu beachten, daß die Division nur den ganzzahligen Teil des Quotienten liefert; ein eventueller Rest wird ignoriert. Alternativen dazu enthält der nächste Abschnitt. Die beiden Worte **D+** und **D−** hinterlassen entsprechend die doppeltgenaue Summe bzw. Differenz von doppeltgenauen Eingangswerten. Die praktische Benutzbarkeit der Postfixnotation in Verbindung mit Kopier- und Ordnungsbefehlen soll an einem Beispiel gezeigt werden. Angenommen, auf dem Stapel liegen zwei Parameter a und b, und es soll der Ausdruck
(a * b)/(a + b) berechnet werden. Das ist mit der folgenden Sequenz möglich:
OVER OVER * ROT ROT + /
Da das nicht besonders anschaulich ist, kann es für Entwicklungszwecke zum Selbstverständnis nützlich sein, in einer Art *vertikalen Notation* die aktuelle Belegung des Stapels nach jedem Kommando als Kommentar festzuhalten (Bild 1.5).

1.4.3. Division mit Rest

Die beiden Worte
/MOD MOD
können benutzt werden, wenn die Vernachlässigung des Restes bei der ganzzahligen Division nicht zulässig ist. **/MOD** liefert dann im TOS den Quotienten und darunter den Rest. **MOD** liefert nur den Rest selbst.

1.4.4. Skalierung

Die beiden Worte
***/ */MOD**
erwarten drei Werte x y z als Parameter und berechnen den Wert von x*y/z, wobei die Zwischenergebnisse intern mit doppelter Genauigkeit geführt werden. */ liefert einen Ergebnisparameter. Mit diesem Operator läßt sich unsere Formel (a*b)/(a+b) aus Abschnitt 1.4.2. auch so berechnen:
OVER OVER + */
Ganz ähnlich wie der eben besprochene Befehl ***/** liefert ***/MOD** ebenfalls das Ergebnis im TOS, dazu aber als zweiten Wert (unter dem TOS) zusätzlich den Rest der abschließenden Division.

1.4.5. Gemischtgenaue Operationen

Gemischtgenaue Operationen sind solche, bei denen Parameter verschiedener Genauigkeit eine Rolle spielen. Zum Beispiel erwartet **UM*** zwei einfachgenaueParameter (vorzeichenlos) und hinterläßt deren doppeltgenaues Produkt (ebenfalls vorzeichenlos). **UM/MOD** erwartet im TOS einen einfachgenauen Divisor und darunter einen doppeltgenauen Dividenden. Das Ergebnis ist wieder einfachgenau; alle Parameter werden als vorzeichenlos aufgefaßt.

1.4.6. Begrenzerbefehle

Die vier Befehle
MAX MIN DMAX DMIN
erwarten je zwei Eingangsparameter und liefern als Ausgangsparameter das Extremum der beiden.

1.4.7. Vorzeichenbefehle

ABS und **DABS** bilden den Betrag von einfach- bzw. doppeltgenauen Zahlen; **NEGATE** und **DENEGATE** bilden jeweils das Zweierkomplement des einfach- bzw. doppeltgenauen Eingangsparameters. Damit wird eine Vorzeichenumkehr erreicht.

1.4.8. *Maschinenahe Operationen*

Die Gruppe

1+ 1– 2+ 2– 2/ D2/

faßt solche Operationen zusammen, die in Programmen erfahrungsgemäß häufig vorkommen und für die gleichzeitig besonders schnelle Realisierungen in Maschinencode möglich sind. Das ist das Erhöhen und das Vermindern um eins und um zwei sowie eine bitweise Rechtsverschiebung des Eingangsparameters (einfach- bzw. doppeltgenau), die in der internen Darstellung eine Division durch zwei realisiert.

wird fortgesetzt

Tafel 1.1 Symbole für die Bedeutung der Stapelparameter

TOS	oberster Wert auf dem Parameterstapel (englisch: Top Of Stack)
+n	Stapeleintrag im Bereich 0<x<=32767
16b	Stapeleintrag mit 16 gültigen Bits
32b	doppeltgenauer Stapeleintrag mit 32 gültigen Bits
8b	Stapeleintrag mit 8 gültigen Bits (b0/ bis b7)
?	Stapeleintrag mit den Werten "true" (? ≠ 0) bzw. "false" (? = 0)
addr	Stapeleintrag, der als Adresse angesehen wird
c	Stapeleintrag, der ein (ASCII-)Zeichen spezifiziert
d	doppelter Stapeleintrag im Bereich −2147483648<x<=2147483647
+d	positive doppelt genaue Zahl 0<x<=4294967295
n	Stapeleintrag im Bereich −32768<x<=32767
u	Stapeleintrag im Bereich 0<x<=65535
ud	doppelter Stapeleintrag im Bereich 0<x<=4294967295
w	Stapeleintrag im Bereich −32768<x<=65535
wb	Stapeleintrag im Bereich −128<x<=255
wd	Stapeleintrag im Bereich −2147483648<x<=4294967295

Zu diesem Kurs

Hier wird dem interessierten Leser die Möglichkeit gegeben, sich über einige der wesentlichen Eigenarten und Potenzen von Forth grob zu orientieren. Wer dadurch angeregt wird, sich ernsthafter mit diesem zukunftsträchtigen Softwarekonzept zu befassen, dem sei das sorgfältige Studium der Bücher von Brodie /1/, /2/ und Zech /3/ empfohlen. Daneben kann man sich in der Spezialliteratur über Themen informieren, die in diesem Kurs nur kurz oder gar nicht erwähnt werden, zum Beispiel: Editieren in Forth / Assemblerprogrammierung / Interrupts, Echtzeit und Multitasking / Metakompilation und Crosscompilation / Fließkommarechnung / Forthprozessor in Hardware.

Hilfreich ist sicherlich auch der Kontakt zu anderen Forthprogrammierern. Für beruflich Interessierte bietet sich hier die Kammer der Technik an. Dort arbeitet in der wissenschaftlichen Sektion Computer- und Mikroprozessortechnik des Fachverbandes Elektrotechnik ein Fachausschuß Forth. Auch außerberuflich an Forth Interessierte haben sich beim Kulturbund in Leipzig zu einer Interessengemeinschaft Forth zusammengefunden, die landesweit aktiv ist.

Tafel 1.2 Kurzbeschreibung der Forthworte

Name	Stapeleffekt	Beschreibung
Zahlenausgaben		
.	(n===>)	16-Bit-Wert mit MSB als Vorzeichen ausgeben
U.	(u===>)	16-Bit-Wert als Positivwert ausgeben
D.	(d===>)	32-Bit-Wert mit MSB als Vorzeichen ausgeben
Parameterstapel; Vervielfältigungen		
DUP	(16b===>16b 16b)	obersten Stapelwert identisch duplizieren
OVER	(16b0 16b1 ===>16b0 16b1 16b0)	zweiten Wert zum TOS kopieren
PICK	(16bj 16bi 16bh ... 16b0 i ===>16bj 16bi 16bh ... 16b0 16bi)	i-ten Wert zum TOS kopieren
?DUP	(0===>0) (16b===>16b 16b)	TOS nur dann duplizieren, wenn er nicht null ist
2DUP	(32b===>32b 32b)	32-Bit Wert duplizieren
2OVER	(32b0 32b1 ===>32b0 32b1 32b0)	zweiten doppeltgenauen Wert zum TOS kopieren
Ordnungsbefehle		
ROT	(16b0 16b1 16b2 ===>16b1 16b2 16b0	dritten Wert nach oben "rollen"
SWAP	(16b0 16b1 ===>16b1 16b0)	obere beide Werte tauschen
ROLL	(16bj 16bi 16bh ... 16b0 i ===>16bj 16bh ... 16b0 16bi)	i-ten Wert nach oben "rollen"
2ROT	(32b0 32b1 32b2) ===>32b1 32b2 32b0)	dritten doppeltgenauen Wert nach oben "rollen"
2SWAP	(32b0 32b1 ===>32b1 32b0)	obere beide doppeltgenauen Werte tauschen
Entfernen von Werten		
DROP	(16b===>)	TOS entfernen
2DROP	(32b===>)	doppeltgenauen Wert vom TOS entfernen
Stapeltiefe		
DEPTH	(===>+n)	Gesamtzahl der 16-Bit-Werte auf dem Stapel
Grundrechenarten		
+	(w1 w2===>w3)	liefert w3 als Summe aus w1 und w2
−	(w1 w2===>w3)	liefert w3 als Diffrenz aus w1 − w2
*	(w1 w2===>w3)	bildet w3 als Produkt aus w1 und w2
/	(n1 n2===>n3)	bildet n3 als Quotient n1/n2
D+	(wd1 wd2===>wd3)	addiert doppeltgenaue Werte wd1 und wd2 zu wd3
D−	(wd1 wd2===>wd3)	subtrahiert zwei doppeltgenaue Zahlen zu wd3
Division mit Rest		
/MOD	(n1 n2===>n3 n4)	bildet Quotient n4 und Rest n3 von n1/n2
MOD	(n1 n2===>n3)	bildet den Rest n3 der Division n1/n2
Skalierung		
*/	(n1 n2 n3===>n4)	n4 = n1*n2/n3; Zwischenergebnis n1*n2 ist doppeltgenau
*/MOD	(n1 n2 n3===>n4 n5)	n5 = n1*n2/n3; n4 ist der Rest bei der Division
Gemischtgenaue Operationen		
UM*	(u1 u2===>ud)	doppeltgenaues Produkt einfachgenauer Positivwerte
UM/MOD	(ud u1===>u2 u3)	Quotient u3 und Rest u2 von ud/u1
Vergleichende Befehle		
MAX	(n1 n2===>n3)	n3 ist die größere der beiden Zahlen n1 und n2
MIN	(n1 n2===>n3)	n3 ist die kleinere der beiden Zahlen n1 und n2
DMAX	(d1 d2===>d3)	d3 ist die größere der doppeltgenauen Zahlen d1 und d2
DMIN	(d1 d2===>d3)	d3 ist die kleinere der doppeltgenauen Zahlen d1 und d2
Vorzeichenbefehle		
ABS	(n===>u)	u ist der Absolutbetrag von n
DABS	(d===>ud)	ud ist der Absolutbetrag von d
NEGATE	(n1===>n2)	n2 ist das Zweierkomplement von n1
DNEGATE	(d1===>d2)	d2 ist das Zweierkomplement von d1
Maschinennahe Operationen		
1+	(w1===>w2)	w1 wird inkrementiert zu w2
1−	(w1===>w2)	w1 wird dekrementiert zu w2
2+	(w1===>w2)	w1 wird um 2 inkrementiert zu w2
2−	(w1===>w2)	w1 wird um 2 dekrementiert zu w2
2/	(w1===>w2)	w1 wird um ein Bit arithmetisch rechtsverschoben
D2/	(wd1===>wd2)	analog 2/ für doppeltgenaue Zahlen

Literatur

/1/ Brodie, L.: Programmieren in Forth. München: Carl Hanser Verlag 1984

/2/ Brodie, L.: In Forth Denken. München: Carl Hanser Verlag 1986

/3/ Zech, R.: Forth-83. München: Franzis Verlag 1987

✉ KONTAKT ☏

WPU Rostock, Sektion Technische Elektronik, WB Automatische Steuerungen, Albert-Einstein-Straße 2, Rostock, 2500; Tel. 452 14



Kurs

Einführung in Forth-83

Teil 2

Dr. Hartmut Pfüller (Leiter),
Dr. Wolfgang Drewelow, Dr. Bernhard Lampe,
Ralf Neuthe, Egmont Woitzel
Wilhelm-Pieck-Universität Rostock,
Sektion Technische Elektronik

2. Kompilieren in Forth

Herkömmliche Compiler sind in sich abgeschlossene Rechenprogramme, die Quelltext so übersetzen, daß das Übersetzungsprodukt (oder auch Kompilat) schließlich in den Speicher gebracht und dort abgearbeitet werden kann. In Forth ist der Übersetzungsvorgang nicht derart in einem separaten Compiler zentralisiert. Hier entsteht das Kompilat durch leicht nachvollziehbare Aktionen des Textinterpreters, wobei sich die zu übersetzenden Worte selbst noch aktiv am Übersetzungsvorgang beteiligen können.
In diesem Teil werden zunächst die Befehle für den Zugriff zum Rechnerspeicher zusammengestellt. Darauf aufbauend folgen Ausführungen zum Übersetzungsvorgang als Wörterbucherweiterung, über Worte zur Programmstrukturierung und zum Umgang mit Teilwörterbüchern.

2.1. Speicherzugriff

2.1.1. Adressen

Für den Übersetzungsprozeß und die Ablage des entstandenen Kompilats sind Zugriffsmöglichkeiten zum Speicher erforderlich. Hierfür benutzt das Forthsystem genau dieselben Worte, die auch für den Programmierer da sind, wenn er auf Speicheradressen zugreifen möchte. Diese Worte erwarten die Speicheradresse, auf die zuzugreifen ist, als 16-Bit-Wert auf dem Parameterstapel. Damit wird die Adressierung von 64 kByte direkt unterstützt.

2.1.2. Übertragung zwischen Speicher und Stapel

Die drei Worte
C@ @ 2@
lesen Werte vom Speicher. Sie erwarten auf dem Stapel eine Adresse, entfernen diese und holen von dem damit adressierten Speicherplatz ein, zwei bzw. vier Byte zum Stapel. Auch beim Befehl C@, der nur ein Byte holt, wird als Ergebnisparameter auf dem Stapel ein 16-Bit-Wert hinterlassen; dabei sind die höherwertigen acht Bit auf Null gesetzt.
Dagegen schreiben die drei Worte
C! ! 2!
auf die im TOS (Top Of Stack) liegende Adresse den darunterliegenden Wert, und zwar auch wieder ein, zwei bzw. vier Byte. Das Wort +! erwartet im TOS ebenfalls eine Speicheradresse. Der darunter liegende Wert wird zum auf der anzusprechenden Speicherzelle liegenden 16-Bit-Wert addiert.

2.1.3. Behandlung von Speicherbereichen

Mit den Worten
CMOVE CMOVE>
kann der Inhalt von Speicherbereichen verschoben werden. CMOVE kopiert dabei zuerst das erste Byte, CMOVE> kopiert zuerst das letzte Byte des Bereichs. CMOVE> eignet sich also zum Aufwärtsverschieben von sich überlappenden Speicherbereichen, was auch durch das Zeichen > am Namen angedeutet werden soll. Das Wort FILL füllt einen Speicherbereich mit dem Byte, das als Parameter im TOS liegt. Größe und Anfangsadresse des Bereichs liegen darunter.

> **Übrigens ...**
>
> ... sollte die im Jahre 1969 von Charles Moore an einem Observatorium in Charlottsville entwickelte Programmiersprache Fourth heißen und zum Ausdruck bringen, daß sie für die 4. Computergeneration konzipiert wurde. 6stellige Bezeichnungen waren aber auf der von Moore genutzten Anlage IBM 1130 noch nicht möglich, so daß er kurzerhand einen Buchstaben wegließ. Damit entstand Forth als Name für eine Echtzeit-Programmiersprache mit einer bis zu zehnmal höheren Geschwindigkeit gegenüber Basic. *MP*

2.2. Die Erweiterung des Wörterbuchs

Befehlsworte werden immer dann nicht ausgeführt, sondern übersetzt, wenn der Kompiliermodus eingeschaltet ist, also zum Beispiel zwischen Doppelpunkt und Semikolon. In diesem Zustand wird das Forthsystem gewissermaßen davon unterrichtet, welche Befehlsabfolge zu einem neu definierten Wort gehören soll. Der Mechanismus, nach dem diese Informationen übersetzt werden, ist denkbar einfach: Nachdem der neue Name im Klartext als nächster Eintrag an das bisherige Ende des Wörterbuchs dazugeschrieben wurde, werden dann der Reihe nach die Codeanfangsadressen der als Programminhalt notierten Worte angehängt. Auf diese Weise wird das Wörterbuch um einen neuen Eintrag erweitert; es wächst.

2.2.1. Wortnamen

Definitionsworte tragen bei ihrer Ausführung einen zusätzlichen Namen in das Wörterbuch ein. Die Form dieser Eintragung im Wörterbuch unterscheidet sich nicht von den Eintragungen, die vorher schon zum Forthsystem gehörten.
Wegen dieser Einheitlichkeit kann das gesamte Forthsystem als ein ausschließlich mit den eigenen Mitteln definiertes Programm angesehen werden. Professionelle Forthsysteme werden auch genau auf diese Weise erzeugt: Sie sind vollständig in Forthquellcode definiert und entstehen als Ergebnis eines Übersetzungsvorgangs mittels Forth. Für die im Standard Forth-83 definierten Funktionen sind möglichst sinnfällige Namen festgelegt. Für den Programmierer gibt es bei der Wahl der Namen für seine eigenen Worte nur die Einschränkung, daß der Name nicht mehr als 31 Zeichen enthalten darf. Sowohl im Ausführungs- als auch im Kompiliermodus ist der nur wenige Zeilen lange Textinterpreter dasjenige Systemprogramm, das die einzelnen Worte aus dem Eingabestrom entgegennimmt und je nach Betriebsmodus aufruft (d. h. ausführt) oder kompiliert. Von den Worten im Quellcode nimmt der Textinterpreter im allgemeinen an, daß sie schon als Einträge im Wörterbuch vorhanden sind. Allerdings sind an dieser Stelle zwei Ausnahmen zu erwähnen: Als erste Ausnahme sind im Eingabestrom Zahlen zulässig, da der Textinterpreter bei unbekannten Worten zunächst versucht, diese als numerische Werte aufzufassen und zu konvertieren. Als zweite Ausnahme kann es geschehen, daß ein vom Textinterpreter aufgerufenes Wort die weitere Analyse des Eingabestroms zeitweilig selbst *in die Hand nimmt*. Dieses Wort kann dann nachfolgende Worte im Quelltext als Parameter werten und eventuell gar nicht erst im Wörterbuch suchen. So arbeiten beispielsweise Definitionsworte, wenn sie für den nachfolgenden Namen einen neuen Wörterbucheintrag einrichten.

2.2.2. Definition von Datenworten

a) Unveränderliche Zahlenwerte

Ein symbolischer Name für einen unveränderlichen Zahlenwert kann mit dem Definitionswort CONSTANT vereinbart werden. CONSTANT erledigt bei seiner Aktivierung zwei Aufgaben: Es schreibt für den neu definierten Wortnamen einen neuen Eintrag ins Wörterbuch und holt den obersten Wert vom Stapel, um ihn im Wörterbuch bei dem neuen Namen abzulegen:

```
26  CONSTANT  BUCHSTABEN(cr)_ok
10  CONSTANT  ZIFFERN(cr)_ok
BUCHSTABEN ZIFFERN + CONSTANT ZEICHEN(cr)_ok
BUCHSTABEN  .(cr)_26_ok
ZIFFERN  .(cr)_10_ok
ZEICHEN  .(cr)_36_ok
```

Der unterstrichene Teil kennzeichnet die Eingabe der Entertaste (cr) und die Rechnerreaktion bis (ok). Nachdem ein Konstantenname definiert wurde, ist die Benutzung dieses Namens funktionell gleichwertig mit der direkten Benutzung der betreffenden Zahl. Hier soll außerdem deutlich werden, daß es nicht etwa syntaktische Vorschrift ist, vor dem Wort CONSTANT eine Zahl zu notieren. Der Programmierer muß nur wissen, daß CONSTANT den augenblicklich auf dem Parameterstapel befindlichen Wert holt und diesen als vereinbarten Konstantenwert ansieht. Wie der Programmierer diesen Wert gewinnt, gehört zur Vorgeschichte und ist für die Funktion von CONSTANT vollständig gleichgültig. Es ist also möglich, den zu vereinbarenden Konstantenwert erst durch ein (beliebig komplexes) Programm ermitteln zu lassen. Weiterhin ist es nützlich zu wissen, daß ein definiertes Datenwort wie zum Beispiel ZIFFERN selbst ein aktives Wort ist. Zu verstehen ist das so, daß das Wort ZIFFERN bei seiner Ausführung völlig selbständig die weitere Programmabarbeitung übernimmt, in eigener Zuständigkeit den Wert 10 zum Para-

meterstapel schafft und anschließend die weitere Programmabarbeitung wieder an das Forthsystem zurückgibt.

b) Veränderliche Zahlenwerte
Symbolische Namen für veränderliche Zahlenwerte werden mit dem Definitionswort VARIABLE vereinbart.

VARIABLE TEILE(cr) ok

Das Wort VARIABLE hat jetzt ans Wörterbuch einen Eintrag mit dem Namen TEILE und dem Code für veränderliche Zahlenwerte angehängt. Dabei wurde eine Speicherzelle freigehalten, in der der jeweils aktuelle Wert von TEILE eingetragen ist. Wenn das Wort TEILE ausgeführt wird, wird automatisch der Codeteil für veränderliche Zahlenwerte aktiviert, und der seinerseits schafft genau die Adresse der freigehaltenen Speicherzelle zum Parameterstapel. Der Programmierer benutzt nun diese Adresse, um mit den gewöhnlichen Worten für den Zugriff auf Speicheradressen den Variablenwert zu setzen oder zu lesen:

```
100  TEILE   !(cr) ok
50  TEILE  +!(cr) ok
TEILE  @  .(cr) 150 ok
-70  TEILE  +!(cr) ok
TEILE  @  .(cr) 80 ok
```

Der Name TEILE ist damit nichts anderes als eine menschliche Merkhilfe für die zugeteilte Speicheradresse. Falls der Programmierer sich dafür interessiert, welche konkrete Speicherzelle das Forthsystem für die Variable TEILE reserviert hat, kann er sich diese Speicheradresse anzeigen lassen.

TEILE .(cr) 32373 ok

Der Variablenname hinterläßt ja bei seiner Ausführung als eigene Aktion die zugeteilte Adresse auf dem Stapel, und der Punkt gibt diesen Wert aus. Ob der Programmierer in dieser angenommenen Situation nun mittels des Namens TEILE oder mittels der Zahl 32373 auf die bewußte Speicherzelle zugreift, ist funktionell gleichwertig. In Quelltexten empfiehlt sich natürlich der Gebrauch des Namens, um in jedem Fall die richtige Bezugnahme zu sichern. Die Verfügbarkeit der Variablenadresse ermöglicht ein flexibles Reagieren auf wechselnde Anforderungen. Angenommen, es gäbe ein bewährtes größeres Programm zur Bestandsverwaltung, mit dem nach dem obigen Muster Zugänge und Abgänge mittels der Variablen TEILE registriert werden. Nun möge sich die Notwendigkeit ergeben, nicht nur wie bisher eine einzige, sondern zwei verschiedene Sorten dieser Teile zu verwalten, beispielsweise solche aus Holz und solche aus Plast. Sicherlich wird man dafür zwei verschiedene Zählvariablen (z. B. #HOLZ und #PLAST) vorsehen müssen. Trotzdem kann bei entsprechend zweckmäßigem Vorgehen gesichert werden, daß der bewährte Programmbestand weitgehend unverändert bleibt. Die Methode besteht darin, das Verhalten von TEILE nachzubilden und bei Bedarf nur die Sorte zu wechseln. Dazu wird eine zusätzliche Variable MATERIAL definiert, in der die jeweils aktuelle Sorte gespeichert ist. Mit den Befehlen HOLZ und PLAST wird dann bei Bedarf die Sorte gewechselt, während die gesamte Zählung weiter mit dem bisherigen Programm für TEILE erledigt werden kann (siehe Bild, vgl. auch Abschnitt 2.4.3.).

```
VARIABLE  #HOLZ        0  #HOLZ   !(cr) ok
VARIABLE  #PLAST       0  #PLAST  !(cr) ok
VARIABLE  MATERIAL       #HOLZ MATERIAL !(cr) ok
: HOLZ    ( ===> )  #HOLZ  MATERIAL ! ;(cr) ok
: PLAST   ( ===> )  #PLAST MATERIAL ! ;(cr) ok
:  TEILE    ( ===> a)  MATERIAL  @  ;(cr) ok
```

Die Variablen haben nach ihrer Definition einen unbestimmten Inhalt und werden deshalb geeignet initialisiert. TEILE ist jetzt keine Variable mehr, sondern ein Programm, das bei Aufruf eine der Variablenadressen #HOLZ oder #PLAST liefert. Wenn das Verwaltungsprogramm aber nur voraussetzt, daß der Aufruf von TEILE die Adresse der Zählvariablen zum Stapel liefert, dann muß es weiter korrekt arbeiten. Für den Benutzer des Programms hat sich nur geändert, daß er jetzt bei Bedarf eine bestimmte Sorte anwählen kann:

```
PLAST  12  TEILE  +!  20  TEILE   +!(cr) ok
HOLZ   65  TEILE  +!  -15  TEILE  +!(cr) ok
PLAST  TEILE  @  .(cr) 32 ok
HOLZ   TEILE  @  .(cr) 50 ok
```

2.2.3. Definition von Programmworten

Die übliche Art und Weise, Forthprogramme zu schreiben, besteht in der Definition von Programmworten mittels des Doppelpunkts. Gute Programmierer finden bei der Problemanalyse heraus, in welche kleinsten funktionellen Einheiten das Problem zerlegt werden kann. Für Forthsoftware ist charakteristisch, daß solche funktionellen Einheiten als sehr kleine eigenständige Programme definiert sind. Ist in einem Programm für elektrische Schaltungen beispielsweise die Parallelschaltung von Widerständen zu berechnen, böte sich an, ein Programm für einen Parallelschaltungsoperator vorzusehen:

: || (r1 r2 ===> r1||r2)(cr) ok
OVER OVER + */ ;(cr) ok

Gesamtwiderstand dreier paralleler Widerstände im Wert von ein, zwei und drei kOhm kann damit folgendermaßen ermittelt werden.

1000 2000 3000 || || .(cr) 545 ok

Der Doppelpunkt als Definitionswort hat den neuen Namen || ins Wörterbuch eingetragen. Nach der Kompilierung der vier Worte, die das Programm ausmachen, schaltet das Semikolon wieder den Ausführungsmodus ein. Mit dem Definieren von Programmen wie || für die elementaren Bestandteile des Problems entsteht nach und nach ein Vorrat an Worten, der seinerseits wieder benutzt werden kann, um die nächsthöheren Niveaus der Aufgabe problemnah zu beschreiben.

2.2.4. Streichen von Definitionen

Im Laufe einer Forthsitzung kann die Lage eintreten, daß das Wörterbuch mit Probierversionen beladen ist, die inzwischen nicht mehr gebraucht werden. Dann gibt es die Möglichkeit, mittels des Wortes FORGET von einem anzugebenden Namen an das gesamte Ende des Wörterbuchs wieder „abzuschneiden". In der folgenden Zeile wird der Eintrag MATERIAL zusammen mit allen nachfolgenden Einträgen aus dem Wörterbuch getilgt.

FORGET MATERIAL(cr) ok

2.3. Programmstrukturierung

Die bisherigen Beispiele enthielten nur reine *Geradeausprogramme*. So trivial sind reale Programme in aller Regel nicht, vielmehr werden Möglichkeiten für Entscheidungen, Verzweigungen und Wiederholungen benötigt.

2.3.1. Logische Werte

Zahlen auf dem Parameterstapel können auch als logische Werte angesehen werden. Dabei gilt in Forth die Vereinbarung, daß die Zahl Null den logischen Wert *falsch* bedeutet. Jede Zahl ungleich Null gilt dagegen als logischer Wert *wahr*. Logische Werte werden im Laufe der Programmabarbeitung zum Beispiel als Ergebnisse von Vergleichsoperationen geliefert. Ein Vergleichsoperator arbeitet dabei nach demselben Schema, wie die arithmetischen Operatoren: Die beiden zu vergleichenden Zahlen liegen oben auf dem Stapel. Der Vergleichsoperator entfernt die Zahlen und hinterlegt auf dem Parameterstapel denjenigen logischen Wert, den er als Ergebnis des Vergleichs herausgefunden hat. Wenn sich der logische Wert *wahr* ergibt, dann erzeugen die Operatoren dafür eine *−1* (alle Bits auf *1* gesetzt). Als Operatorname für den Test auf Gleichheit wird naheliegenderweise das Gleichheitszeichen = benutzt. Ähnlich selbsterklärend wirken die Operatoren für *größer als* und *kleiner als*, wie das Bild zeigt.

```
1   2   =   .(cr) 0 ok
3   3   =   .(cr) -1 ok
-5  0   <   .(cr) -1 ok
-5  0   >   .(cr) 0 ok
```

Häufig tritt der Fall auf, daß eine Zahl mit Null verglichen werden soll. Vom Standard sind für diese Fälle drei Operatoren (**0=** **0>** und **0<**) vorgesehen. Alle bisher genannten Vergleichsoperatoren sehen die zu vergleichenden Zahlen als vorzeichenbehaftete 16-Bit-Zahlen an. Wenn die zu vergleichenden Zahlen als vorzeichenlos angesehen werden sollen, muß der Operator U< benutzt werden.

30000 40000 U< .(cr) −1 ok

Für den Vergleich von doppeltgenauen Zahlen sieht der Standard die folgenden vier Operatoren vor:

D0= D= D< DU<

2.3.2. Bitoperationen

Zur bitweisen Verknüpfung von zwei logischen Eingangswerten zu einem logischen Ergebniswert dienen die Worte

AND OR XOR

Das Wort NOT bildet die bitweise Negation (Einerkomplement) eines Eingangswertes.

2.3.3. Entscheidungsstrukturen

Logische Werte (z. B. aus Ergebnissen von Vergleichen) werden häufig zur Entscheidung über Programmverzweigungen benutzt. In Forthprogrammen wird die Verzweigung mittels der Worte IF ELSE und THEN organisiert:

.... IF .... ELSE .... THEN ....

Die Syntax unterscheidet sich völlig von der

in herkömmlichen Programmiersprachen, ist aber in sich wieder sehr logisch: Bereits *vor* dem IF muß der Programmierer alle erforderlichen logischen Operationen erledigt haben, also zum Beispiel Vergleiche, logische Verknüpfungen ihrer Ergebnisse usw. Das Ergebnis liegt nun als logischer Wert im Sinne einer Steuerflagge (englisch: flag) auf dem Parameterstapel. Das Wort IF holt diesen logischen Wert vom Stapel, wertet ihn aus, und organisiert, wo der Programmlauf weiterzugehen hat. In diesem Sinne kann IF als Verzweigungsoperator angesehen werden, der entsprechend der Postfixnotation hinter (!) den als Steuerflagge dienenden Operanden steht. IF selbst leitet dann schon den Ja-Zweig ein und ist deshalb in der deutschen Entsprechung als **falls-ja** zu verstehen. Der Nein-Zweig beginnt mit ELSE, das in der deutschen Entsprechung als **andernfalls:** zu verstehen ist. Beendet wird die Verzweigung mit dem Wort THEN im Sinne des deutschen **danach:** Mit diesen Mitteln der Verzweigung kann beispielsweise ein Wort L. für die **freundlichere** Anzeige von logischen Werten programmiert werden:

```
:  L.          ( flag ===> )(cr)_ok
   IF     ." wahr "            ( falls ja)(cr)_ok
   ELSE   ." falsch "     ( andernfalls)(cr)_ok
   THEN  ;(cr)_ok
36  ZEICHEN  =  L.(cr)_wahr_ok
TEILE  @  0=  L.(cr)_falsch_ok
```

Der ELSE-Teil kann auch weggelassen werden:

```
:  KONTROLLE     ( ===> )(cr)_ok
   TEILE  @  0<(cr)_ok
     IF     ."  Achtung, Defizit! "(cr)_ok
     THEN  ;(cr)_ok
-90  TEILE  +!(cr)_ok
KONTROLLE(cr)__Achtung,_Defizit!_ok
```

2.3.4. Schleifen mit unbestimmter Durchlaufzahl

Wenn eine Programmpassage mehrmals abgearbeitet werden soll, kann sie zwischen die Worte BEGIN und UNTIL geschrieben werden:

.... BEGIN .... UNTIL ....

Bei der Abarbeitung des Programms konsumiert UNTIL in jedem Durchlauf einen Wert vom Parameterstapel und lenkt den Programmfluß immer dann um (zu BEGIN zurück), wenn der geholte Wert gleich Null (also *falsch*) war. Wenn UNTIL einen Wert ungleich Null (also *wahr*) auf dem Stapel vorfindet, wird der Programmfluß nicht umgelenkt, sondern hinter UNTIL fortgesetzt; damit wird die Schleife verlassen. Den Wert, den UNTIL vom Stapel abholt, muß der Programmierer durch geeignete Gestaltung des Programms bereitstellen. Im folgenden Beispiel wird von einer vorzugebenden Speicheradresse an so lange Byte für Byte ausgelesen und angezeigt, bis ein Speicherplatz mit dem Inhalt Null gefunden wird.

```
:  INHALT      ( addr ===> )
     BEGIN     ( addr                   )
        DUP    ( addr    addr           )
        1+     ( addr    addr+1         )
        SWAP   ( addr+1 addr            )
        C@     ( addr+1 byte            )
        DUP    ( addr+1 byte     byte)
        .      ( addr+1 byte            )
        0=     ( addr+1 flag            )
     UNTIL     ( addr+1                 )
   DROP  ;     (                        )
```

Der hier noch einmal besonders ausführliche Stapelkommentar wird in späteren Beispielen weggelassen, sollte aber vom Leser im Zweifelsfall nachgeholt werden. Zuweilen wird eine Schleife gebraucht, wo über die Wiederholung nicht erst am Ende, sondern schon früher entschieden wird. Dafür kann die Konstruktion

... BEGIN ... WHILE ... REPEAT ...

benutzt werden. Von REPEAT wird der Programmfluß unbedingt zurück zu BEGIN gelenkt. WHILE ist dasjenige Wort, das vom Parameterstapel eine Steuerflagge holt und auswertet. Bei einem Wert ungleich Null (*wahr*) wird der Programmfluß nicht umgelenkt, sondern hinter WHILE fortgesetzt; durch REPEAT wird dann die Schleife wiederholt. Bei einem Wert gleich Null (*falsch*) wird durch WHILE der Programmfluß zu der Stelle hinter REPEAT umgelenkt; die Schleife wird also verlassen. Mit dieser Art Schleife kann das obige Programm so modifiziert werden, daß die abschließende Null nicht mit ausgegeben wird:

```
:  INHALT1      ( addr ===> )(cr)_ok
      BEGIN    DUP   C@   DUP(cr)_ok
      WHILE    .   1+(cr)_ok
      REPEAT(cr)_ok
   2DROP  ;(cr)_ok
```

2.3.5. Zählschleifen

Zählschleifen verwalten einen sogenannten *Laufindex*, für den zwei Grenzwerte vorgegeben werden. Nach jedem Schleifenzyklus wird zum Laufindex ein gewisser Wert, das *Inkrement*, addiert. Mit dem Inkrement eins arbeitet die Zählschleife

... DO .... LOOP ....

Das Wort DO erwartet auf dem Stapel den Startwert des Laufindex, und darunter dasjenige Limit, für das die Schleife *nicht* mehr abgearbeitet werden soll. LOOP erhöht nach jedem Zyklus den Laufindex um eins und prüft, ob das Limit erreicht ist. Wenn das Limit noch nicht erreicht ist, wird die Abarbeitung hinter DO wiederholt; ansonsten wird die Schleife verlassen und das Programm hinter LOOP fortgesetzt. Das folgende Bild zeigt

```
:  STERNE       ( n ===> )(cr)_ok
   0  DO     ." *"(cr)_ok
      LOOP  ;(cr)_ok
5  STERNE(cr)_*****ok
```

das Benutzungsprinzip. Das DO findet oben auf dem Stapel als Anfangswert für den Laufindex eine Null und darunter das Limit. Daß diese Werte aus verschiedenen Quellen stammen, nämlich das Limit vom Dialog des Programmierens und die Null vom Programm STERNE, gehört für DO zur Vorgeschichte und ist damit nicht interessant. Wichtig ist nur das Vorhandensein der beiden Werte auf dem Stapel beim Arbeitsbeginn von DO. Wenn der Programmierer innerhalb der Schleife den aktuellen Wert des Laufindex benötigt, kann das Wort I benutzt werden, das diesen Wert auf dem Stapel hinterlegt. Mit diesen Mitteln kann ein kleines DUMP-Programm geschrieben werden:

```
:  WDUMP      ( anfang ende ===> )(cr)_ok
   SWAP  DO    I  @  .   2(cr)_ok
         +LOOP  ;(cr)_ok
```

Andere Inkremente als Eins (auch negative!) sind verwendbar, wenn die Schleife mit +LOOP abgeschlossen wird. + LOOP holt in jedem Durchlauf das aktuelle Inkrement als Zahlenwert vom Stapel und addiert es zum Index. Die Schleife wird nicht mehr wiederholt, wenn der Index die Grenze zwischen Limit und Limit-1 überschritten hat. Im folgenden Bild wird +LOOP in einem DUMP-Programm für 16-Bit-Zellen benutzt.

```
:  DUMP        ( anfang ende ===> )(cr)_ok
   SWAP  DO    I   C@   .(cr)_ok
         LOOP   ;(cr)_ok
```

Entsprechend den Regeln der strukturierten Programmierung sind alle angeführten Programmstrukturen schachtelbar. Für ineinander geschachtelte Zählschleifen kann es wünschenswert sein, innerhalb der inneren Schleife den aktuellen Wert des Laufindex der nächstäußeren Schleife festzustellen. Der Standard sieht dafür das Wort J vor. Schließlich gibt es noch die Möglichkeit, Zählschleifen vorzeitig zu verlassen. Zu diesem Zweck wird das Wort LEAVE benutzt. Wenn LEAVE erreicht wird, verzweigt der Programmfluß direkt zu der Stelle hinter LOOP (bzw. +LOOP). Um das zu demonstrieren, wird das obige DUMP-Programm so geändert, daß es vorzeitig endet, falls es auf ein Byte gleich Null trifft:

```
:  DUMP1      ( anfang ende ===> )(cr)_ok
   SWAP  DO  I   C@   DUP   .   0=(cr)_ok
                 IF    DROP  LEAVE(cr)_ok
                 THEN(cr)_ok
         LOOP   ;(cr)_ok
```

2.4. Die Gliederung des Wörterbuchs

2.4.1. Vokabulare

Forth bietet die Möglichkeit, inhaltlich zusammengehörige Worte in spezialisierten, eigenständigen Teilwörterbüchern, den sogenannten Vokabularen, zusammenzufassen. Jedes Vokabular erhält dabei einen eigenen Namen. Mit dem Definitionswort VOCABULARY kann der Programmierer Namen für neue Vokabulare vereinbaren. Alle vom Standard vorgesehenen Forthworte befinden sich in einem Vokabular mit dem Namen FORTH.

2.4.2. Die Suchordnung

Das Vorhandensein mehrerer Vokabulare macht Festlegungen darüber erforderlich, welche Vokabulare in welcher Reihenfolge durchsucht werden sollen. Dies wird vom Standard Forth-83 gegenwärtig noch nicht verbindlich geregelt. Nachfolgend wird ein inzwischen verbreitetes Verfahren beschrieben, das im Standard als Vorschlag enthalten ist. Um den Suchraum festzulegen, werden ein oder mehrere Vokabulare als *resident* und genau ein einziges als *transient* erklärt. Die Namen aller Worte im transienten Vokabular kann man sich anzeigen lassen, indem man WORDS aufruft. Die Suche beginnt grundsätzlich im transienten Vokabular und wird danach in den residenten Vokabularen fortgesetzt. Das Aufrufen eines Vokabularnamens führt dazu, daß es zum transienten Vokabular und damit zum ersten in der

Suchreihenfolge wird. Das Installieren einer gewünschten Suchordnung wird meist mit ONLY FORTH eingeleitet. ONLY entfernt sämtliche bisherigen Vokabulare aus dem Suchraum und erklärt ein spezielles Minimalvokabular als resident und gleichzeitig als transient. Durch das Nennen des Vokabulars FORTH wird es zum neuen transienten Vokabular. ALSO kopiert das transiente Vokabular zusätzlich in die Gruppe der residenten Vokabulare. Durch Nennen eines weiteren Vokabulars würde dieses nun zum transienten, könnte dann ebenfalls mit ALSO in die Gruppe der residenten übernommen werden usw. Mit dem Vokabularmechanismus kann man z. B. organisieren, daß Nutzern nur die Worte zugänglich sind, die ihrer Qualifikation entsprechen. Als zusätzliche Sicherheit (z. B. nach Fertigstellung eines Anwendungsprogramms) kann mit dem Wort SEAL (deutsch: versiegeln) die Suchreihenfolge eingefroren und der Umwählmechanismus unzugänglich gemacht werden.

2.4.3. Das Anfügevokabular

In das Anfügevokabular werden neu zu definierende Worte eingetragen. Um ein ganz bestimmtes Vokabular zum Anfügevokabular zu machen, muß es zunächst als transient erklärt werden. Ein Aufruf von DEFINITIONS bewirkt dann, daß das transiente Vokabular auch Anfügevokabular wird. Die Anzeige derjenigen Vokabulare, die aktuell zum Suchen und Anfügen vorgesehen sind, wird durch Aufruf des Wortes ORDER erreicht. Das folgende Bild zeigt als Variation zum Problem im Abschnitt 2.2.2. b), wie mittels der Vokabulartechnik zwei verschiedene Variablen mit dem gleichen Namen TEILE getrennt behandelt werden können.

```
ONLY  FORTH  DEFINITIONS(cr)_ok
VOCABULARY  HOLZ(cr)_ok
VOCABULARY  PLAST(cr)_ok
ALSO  HOLZ  DEFINITIONS(cr)_ok
VARIABLE  TEILE      0  TEILE  !(cr)_ok
PLAST  DEFINITIONS(cr)_ok
VARIABLE  TEILE      0  TEILE  !(cr)_ok
22  TEILE  +!  HOLZ  33  TEILE  +!(cr)_ok
PLAST  TEILE  @  .(cr)_22_ok
HOLZ   TEILE  @  .(cr)_33_ok
```

wird fortgesetzt

Tafel 2.1 Kurzbeschreibungen der Forthworte

Name	Stapeleffekt	Beschreibung
Speicherzugriff: Transfer zwischen Speicher und Stapel		
!	(16b addr ===>)	16b auf Adresse addr abspeichern,
@	(addr ===> 16b)	16b aus der Speicherzelle addr lesen
C!	(8b addr ===>)	8b auf Adresse addr abspeichern
C@	(addr ===> 8b)	8b aus der Speicherzelle addr lesen
+!	(w addr ===>)	zum Inhalt der Speicherzelle addr wird w addiert
2@	(addr ===> 32b)	32b ab Speicherzelle addr lesen
2!	(32b addr ===>)	32 ab Speicherzelle addr abspeichern
Behandlung von Speicherbereichen		
CMOVE	(addr1 addr2 u ===>)	ab Adresse addr1 u Bytes nach addr2 transportieren
CMOVE>	(addr1 addr2 u ===>)	ähnlich CMOVE, aber beginnend bei Adresse addr1+u−1
FILL	(addr u 8b ===>)	ab Adresse addr u Bytes mit dem Bitmuster 8b füllen
Bitoperationen		
AND	(16b1 16b2 ===> 16b3)	bitweise UND-Verknüpfung von 16b1 und 16b2 ergibt 16b3
NOT	(16b1 ===> 16b2)	16b2 ist das Einerkomplement von 16b1
OR	(16b1 16b2 ===> 16b3)	bitweise ODER-Verknüpfung von 16b1 und 16b2 ergibt 16b3
XOR	(16b1 16b2 ===> 16b3)	bitweise XOR-Verknüpfung von 16b1 und 16b2 ergibt 16b3
Definition von Worten		
:	(===>)	Beginn der Definition eines Hochwortes
;	(===>)	Beendigung der Definition eines Hochwortes
CONSTANT	(n ===>)	Konstante definieren
VARIABLE	(===>)	Variable definieren
Wörterbucharbeit		
FORGET	(===>)	Abtrennen des Wörterbuches ab folgendem Namen
Vergleiche		
0<	(n ===> ?)	flag = true, wenn Bedingung erfüllt ist
0>	(n ===> ?)	
0=	(n ===> ?)	
<	(n1 n2 ===> ?)	
>	(n1 n2 ===> ?)	
=	(n1 n2 ===> ?)	
U<	(u1 u2 ===> ?)	
D0=	(d ===> ?)	
D=	(d1 d2 ===> ?)	
D<	(d1 d2 ===> ?)	
DU<	(ud1 ud2 ===> ?)	
Entscheidungsstrukturen		
IF	(? ===>)	Anfang des *Wahr*-Zweiges
ELSE	(===>)	Anfang des *Falsch*-Zweiges
THEN	(===>)	Ende der Verzweigungskonstruktion
Schleifen mit unbestimmter Durchlaufzahl		
BEGIN	(===>)	Anfang der Schleife
UNTIL	(? ===>)	Verlassen, wenn ? = wahr
WHILE	(? ===>)	Verlassen, wenn ? = falsch
REPEAT	(===>)	Rückkehr zu BEGIN
Zählschleifen		
DO	(limit anfangswert ===>)	Eintritt in die Schleife
LOOP	(===>)	Schrittweite = 1
+LOOP	(schrittweite ===>)	Schrittweite variabel
I	(===> n)	Index der inneren Schleife
J	(===> n)	Index der äußeren Schleife, falls sie existiert
Vokabulare		
VOCABULARY	(===>)	Definitionswort für ein neues Vokabular
FORTH	(===>)	Vokabular FORTH als transient einstellen
WORDS	(===>)	Inhalt des transienten Vokabulars anzeigen
ONLY	(===>)	Suchordnung leeren
ALSO	(===>)	das transiente Vokabular wird zusätzlich resident.
SEAL	(===>)	*Versiegeln* der Suchordnung
DEFINITIONS	(===>)	das transiente Vokabular wird auch Anfügevokabular
ORDER	(===>)	Anzeigen von Suchordnung und Anfügevokabular

Einführung in Forth-83

Teil 3

Dr. Hartmut Pfüller (Leiter),
Dr. Wolfgang Drewelow, Dr. Bernhard Lampe,
Ralf Neuthe, Egmont Woitzel
Wilhelm-Pieck-Universität Rostock,
Sektion Technische Elektronik

3. Das Forth-Betriebssystem

Forth wird vielfach nicht nur als Programmiersprache, sondern als Betriebssystem, als Programmierwerkzeug oder als Softwareumgebung bezeichnet. Ein Grund dafür ist sicherlich, daß die Programmierung nicht wie bei den meisten Sprachen üblich in der Reihenfolge Editor, Compiler, Debugger mit der immer dazwischenstehenden Stufe Betriebssystem abläuft, sondern einheitlich in der Forthumgebung erfolgt (eventuell auch unter Nutzung von äquivalenten Forthkomponenten). Ein anderer Grund liegt darin, daß Forth bereits im Kern eine Schichtenstruktur besitzt, die stark an den Aufbau einfacher Betriebssysteme angelehnt ist: In der untersten Schicht realisiert Forth auf einem sehr niedrigen Niveau den Bezug zur Standardperipherie (Tastatur, Bildschirm, Massenspeicher). In der darüberliegenden Schicht arbeiten dann schon geräteunabhängige Worte zur Zeichenkettenverarbeitung und -konvertierung sowie Funktionen des virtuellen Massenspeicherkonzepts. In einer weiteren Schicht erfolgt die Definition des Kommandointerpreters des Systems.
Die Schnittstelle zwischen dem Forthsystem und der Peripherie kann sehr schmal gehalten werden und ermöglicht deshalb eine schnelle Implementierung von Forth auch auf der nackten Hardware. Forth kann damit auch als kompaktes und sehr flexibles Betriebssystem auf beliebigen, aufgabenspezifisch zugeschnittenen Rechnerkonfigurationen eingesetzt werden und bietet darüber hinaus (im Unterschied zu anderen kleineren Betriebssystemen) den gesamten eigenen Sprachumfang für die Programmentwicklung.

3.1. Konsolfunktionen

3.1.1. Zeichenweise Ein- und Ausgaben

Die beiden grundlegenden Worte für die Konsolbedienung sind die Einzelzeichentreiber KEY und EMIT. Die Funktion **KEY** wartet auf das nächste über die Tastatur eingegebene Zeichen und legt es als 16-Bit-Wert auf den Datenstapel. In den unteren 7 Bits befindet sich die ASCII-Information. Darüber hinaus werden aber alle Bits entgegengenommen – die höherwertigen Bits können systembedingt auch ungleich Null sein. Die zu KEY komplementäre Aktion **EMIT** gibt das in den niederwertigen 7 Bits befindliche ASCII-Zeichen einer auf dem Stapel liegenden Zahl an das Terminalgerät aus. Die höherwertigen Bits können wieder gerätespezifisch angewandt werden. Mit Hilfe von EMIT ist die Definition von weiteren zum Kernwortschatz gehörenden Konsolfunktionen möglich. Das Wort **SPACE** zur Ausgabe eines Leerzeichens (ASCII-Code 32) könnte im Forthsystem so definiert sein:

```
32  CONSTANT  BL

:  SPACE     ( ===> )
   BL  EMIT  ;
```

Die Ausgabe von mehreren ASCII-Leerzeichen kann durch die Verwendung von SPACES erfolgen, wobei auf dem Stapel die Anzahl erwartet wird. Eine Definition von SPACES wäre unter Verwendung von SPACE innerhalb einer **DO-LOOP**-Konstruktion möglich:

```
:  SPACES     ( +n ===> )
   0  MAX  ?DUP  IF  0  DO  SPACE
                            LOOP
                 THEN  ;

:  CR         ( ===> )
   13  EMIT  10  EMIT  ;
```

Das Wort **CR** führt einen Wagenrücklauf und Zeilenvorschub (Carriage Return – Linefeed, ASCII-Codes 13 und 10) aus.
Eine häufig implementierte, allerdings nicht im Standard fixierte Funktion ist **KEY?**. Dieses Wort testet den aktuellen Status des Eingabegerätes: Liegt eine Tastenbetätigung vor, so wird ein True-Flag auf den Stapel übergeben, andernfalls ein False-Flag.

3.1.2. Zeichenkettenausgabe

Eine Forthzeichenkette (Forthstring) umfaßt in ihrer Standarddarstellung zwei Informationsanteile: die Längeninformation und die eigentliche Zeichenfolge. Die Längeninformation befindet sich im ersten Byte (Zählbyte). Danach schließt sich die Zeichenfolge an:

```
100 101 102 103 ...       ---->    aufsteigende Adreßrichtung

:   :   :   :   :   :   :   :   :   :   :   :   :   :   :   :   :   :   :
:18 : D : a : s :   : i : s : t :   : e : i : n :   : S : t : r : i : n : g :
:___:___:___:___:___:___:___:___:___:___:___:___:___:___:___:___:___:___:___:
  ^
  :                          ASCII-Zeichen
Zählbyte
(englisch: count)
```

Erfordert ein Programm eine getrennte Behandlung der Längeninformation und der Zeichenfolge, so kann das Wort **COUNT** benutzt werden. COUNT erwartet auf dem Stapel die Anfangsadresse einer Zeichenkette und erzeugt daraus zwei Ergebnisparameter: Auf der Stapelspitze liegt der Inhalt des Zählbytes, darunter die Adresse des ersten Zeichens der Zeichenkette. Das Wort **TYPE** baut auf den beiden von COUNT gelieferten Parametern auf und bringt die Zeichenkette zur Anzeige. COUNT und TYPE werden deshalb häufig im Zusammenhang benutzt:

```
100  COUNT  TYPE (cr)_Das_ist_ein_String_ok
```

Das Wort **-TRAILING** geht von den gleichen Ausgangsparametern wie TYPE aus. Es prüft, ob das Ende der Zeichenkette aus Leerzeichen besteht, die bei der Weiterverarbeitung der Kette weggelassen werden können. Die auf dem Stapel liegende Adresse des ersten Zeichens bleibt unverändert, nur das Zählbyte wird gegebenenfalls verkleinert. Die Befehlsfolge
addr COUNT -TRAILING TYPE
sorgt also gegenüber der Folge
addr COUNT TYPE
für eine Verkürzung der Ausgabeaktion, sofern am Stringende Leerzeichen standen. Soll innerhalb eines Forthprogramms die Ausgabe einer festen Zeichenkette erfolgen, so kann dazu das Wort ." benutzt werden. Die Zeichenfolge **ccc** der Sequenz ." **ccc**" wird innerhalb einer Wortdefinition in eine Forthzeichenkette umgewandelt und bei der späteren Ausführung dieses Wortes ausgegeben (die Zeichen zwischen den Anführungsstrichen, aber ohne das dem Wort ." folgende Leerzeichen):

```
:  TREFFER    ( n ===> )
   ." Das waren "  .   ." Treffer ! "  ;

5  TREFFER(cr)_Das_waren_5_Treffer_!_ok
```

3.1.3. Kettenorientierte Eingabe

Ein zentrales Wort für die Zeichenketteneingabe ist **EXPECT**. Es wird insbesondere auch vom Textinterpreter benutzt. Eingangsparameter für EXPECT sind die Adresse, auf der die geholte Zeichenkette im Speicher abgelegt werden soll (Adresse des ersten Zeichens, nicht Adresse der ersten Zählbytes!), und eine Längenvorgabe. Die Zeichenketteneingabe wird durch **(cr)** oder durch das Erreichen der Längenvorgabe beendet. Durch EXPECT wird das Zählbyte der Zeichenkette nicht gesetzt. Dafür steht die Anzahl der eingegebenen Zeichen in der Systemvariablen **SPAN**. Nachfolgend ist ein Beispiel für die Anwendung von EXPECT und SPAN zur Umwandlung einer über die Tastatur eingegebenen Folge von Zeichen (maximal 20) in eine Forthzeichenkette im Standardformat angegeben.

```
:  GET_STRING      ( addr ===> )
   DUP  1+  20  EXPECT
   SPAN  @  SWAP  C!  ;
```

Das Wort **GET_STRING** erwartet dabei als Eingangsparameter die Zieladresse des Strings. Nachdem mit EXPECT die Zeicheninformation auf die Zieladresse + 1 abgelegt wurde, wird das Zählbyte unter Nutzung der in SPAN stehenden Längeninformation gesetzt. Die Benutzung von GET_STRING zeigt folgendes Beispiel:

```
VARIABLE  KETTE       19  ALLOT(cr)_ok
KETTE  GET_STRING  (cr)_Hallo, Paul !  ok
KETTE  COUNT  TYPE  (cr)_Hallo,_Paul_!_ok
```

Der Ausdruck **19 ALLOT** nach der Variablendefinition erweitert den Speicherplatz für die Variable um 19 zusätzliche Bytes.

3.2. Zahlenkonvertierung

Im Forthsystem sind Funktionen zur Ein- und Ausgabekonvertierung von Ganzzahlen implementiert. Forth bietet die einzelnen Bestandteile dieser Funktionen auch nach außen an und erlaubt damit dem Anwender, eigene, modifizierte Konvertierungen zu entwerfen.

3.2.1. Steuern der Zahlenbasis
Zentraler Bezugspunkt für die Ein- und Ausgabekonvertierung ist die aktuelle Zahlenbasis. Der Wert dieser aktuellen Zahlenbasis ist in der Systemvariablen **BASE** gespeichert. Eine Veränderung von BASE wirkt damit unmittelbar auf die Konvertierung. Die dezimale Zahlenbasis wird durch **DECIMAL** eingestellt, was intern wie folgt definiert sein könnte:

```
:  DECIMAL      ( ===> )
   10  BASE  !  ;
```

Analog können auch andere Worte zum Einstellen der Zahlenbasis vereinbart werden:

```
DECIMAL

 :  HEX          ( ===> )
    16  BASE  !  ;

 :  OCTAL        ( ===> )
    8  BASE  !  ;

 :  BINARY       ( ===> )
    2  BASE  !  ;
```

Die Nutzung dieser Möglichkeit soll hier anhand der Erzeugung einer Tabelle für unterschiedliche Zahlendarstellungen demonstriert werden. Das Wort **TABELLE** stellt in jedem Schleifendurchlauf einen Zählwert als Dezimal-, Oktal-, Hexadezimal- und Binärzahl dar:

```
:  TABELLE      ( n ===> )
   BASE  @  SWAP
   1  DO  CR  HEX       I  .
              DECIMAL   I  .
              OCTAL     I  .
              BINARY    I  .
      LOOP
   BASE  !  ;
```

Die Tabellenlänge wird dabei als Parameter auf dem Stapel übergeben. Die Rahmenaktionen **BASE @ SWAP** und **BASE !** sorgen dafür, daß zunächst der Wert der Zahlenbasis auf dem Stapel abgelegt wird und daß zum Abschluß dieser noch immer auf dem Stapel befindliche Wert wieder in die Variable BASE eingetragen wird. Ohne diesen Rahmen wäre nach der Abarbeitung von TABELLE immer die zuletzt benutzte binäre Zahlenbasis eingestellt.

3.2.2. Ausgabekonvertierung
Die Ausgabekonvertierung arbeitet nach folgendem Prinzip: Zunächst wird die auf dem Stapel liegende Zahl gegebenenfalls zu einer 32-Bit-Zahl ergänzt. Danach wird wiederholt das Wort # aufgerufen. Damit wird die Zahl jedesmal durch die aktuelle Zahlenbasis dividiert, bis der Quotient 0 ist. Die sich jeweils ergebenden Reste (die zwischen 0 und dem Wert der Zahlenbasis minus 1 liegen) werden in die entsprechenden ASCII-Ziffernzeichen umgewandelt und in der Reihenfolge von der kleinsten bis zur höchsten Stelle (von rechts nach links) in einem Stringzwischenspeicher abgelegt. Nach Beendigung der eigentlichen Wandlung kann der fertige String ausgegeben werden. Der Aufbau des Zahlenstrings wird durch die Operation <# initialisiert. Die eigentliche Konvertierung darf erst nach Abarbeitung dieser Funktion beginnen. Die Sammeloperation #S führt die Funktion # gleich so oft nacheinander aus, bis der 32-Bit-Quotient 0 ist, also bis die gesamte Zahl zu Ende gewandelt ist. Die doppelt genaue Null (letzter Quotient) bleibt nach der Operation #S noch auf dem Stapel liegen, damit die gleichen Stapelbedingungen wie bei der Anwendung von # gesichert werden. Der Abschluß der numerischen Ausgabekonvertierung erfolgt durch #>, wobei auf dem Stapel der letzte Quotient erwartet wird. Ausgangsparameter sind die Adresse des ersten Stringzwischenspeichers und das Zählbyte auf der Stapelspitze. Damit kann unmittelbar nach #>das Wort TYPE angewendet werden. Die interne Definition der Funktion **U.** zur Ausgabe einer vorzeichenlosen ganzen Zahl könnte damit so aussehen:

```
:  U.        ( n ===> )
   0         ( Ergaenzung auf 32 Bit)
   <#  #S  #>  TYPE  SPACE  ;
```

Möchte man zur Strukturierung der Ausgabe an einer bestimmten Stelle in der Ergebnisdarstellung zusätzlich ein ASCII-Zeichen (zum Beispiel einen Dezimalpunkt) unterbringen, so kann dies durch Ausführung der Funktion **HOLD** an der entsprechenden Stelle des Konvertierungsvorgangs erfolgen. HOLD erwartet als Eingangsparameter den ASCII-Code des einzufügenden Zeichens und hinterläßt keinen Ausgangsparameter. Eine strukturierte Ausgabe unter Verwendung von HOLD kann beispielsweise bei der Anzeige der Uhrzeit benutzt werden. Soll für eine auf dem Stapel liegende Sekundenzahl eine Zeitanzeige in der Form Std:Min:Sek erfolgen, so ist das unter Benutzung der Definitionen möglich:

```
:  SEXTAL       ( ===> )
   6  BASE  !  ;

:  :XX          ( n1 ===> n2)
   DECIMAL  #  SEXTAL  #  58  HOLD  ;

:  .ZEIT        ( n ===> )
   BASE  @  SWAP
   0  <#  :XX  :XX  DECIMAL  #S  #>
   TYPE  SPACE  BASE  !  ;


7572  .ZEIT  (cr) 2:06:12 ok
```

Das Wort :**XX** dient der Darstellung der Sekunden bzw. Minuten, die jeweils im Bereich von 0 bis 59 liegen. Hier erfolgt zunächst die Konvertierung der rechten Stelle, die zwischen 0 und 9 liegt (dezimale Basis), dann die Konvertierung der linken Stelle (Bereich 0 bis 5, sextale Basis). Abschließend wird „links" mit **58 HOLD** ein Doppelpunkt erzeugt.

In **.ZEIT** wird zunächst die Ergänzung auf 32 Bit vorgenommen. Danach erfolgt die Ausgabe der Sekunden und Minuten durch das Wort :**XX** und mit **#S** die Wandlung des Restes, das heißt der Stunden. Abschließend wird der erzeugte String mit TYPE ausgegeben.

Das Wort **SIGN** dient der Vorzeichenbehandlung bei der Ausgabekonvertierung. Es fügt den ASCII-Code eines Minuszeichens in den Zahlenstring ein, falls der auf dem Stapel liegende Wert negativ ist. SIGN wird zumeist direkt vor #> angewendet (das heißt, am Ende des Konvertierungsvorgangs). Es plaziert damit gegebenenfalls ein Minuszeichen an die am weitesten links stehende Stelle des Zahlenstrings (vor die Ziffern). Bei Konvertierungsaktionen wird als Zeichenkettenzwischenspeicher vielfach ein dem Anwender zugänglicher Speicher benutzt, dessen Startadresse durch das Wort **PAD** geliefert wird. Allerdings sind Forthworte, die unter Benutzung von PAD geschrieben wurden, im allgemeinen nicht wiedereintrittsfähig.

Mit den oben beschriebenen Teilfunktionen zur Ausgabekonvertierung lassen sich (wie am Beispiel von U. beschrieben) sämtliche in Forth vorhandenen Worte zur Zahlenausgabe definieren. In vielen Implementierungen von Forth-83 sind die Worte **.R** und **D.R** vorhanden. Diese Funktionen nehmen die rechtsbündige Ausgabe einer einfach- bzw. doppeltgenauen Integerzahl vor, wobei eine auf der Stapelspitze stehende positive Zahl die Weite des Ausgabefeldes festlegt.

3.2.3. Eingabekonvertierung
Die vom Textinterpreter benutzte zentrale Funktion zur Eingabekonvertierung ist
CONVERT (+d1 addr1 => +d2 addr2)
Von der ab adr1 + 1 stehenden Zeichenkette wird das erste ASCII-Zeichen geholt. Sofern dieses Zeichen ein gültiges Ziffernsymbol entsprechend der eingestellten Zahlenbasis ist, wird die konvertierte Ziffer zum doppelt genauen Produkt aus Zahlenbasis und 32-Bit-Stapelwert addiert. Die Adresse innerhalb des Zahlenstrings wird erhöht. Dieses Verfahren wird solange fortgesetzt, bis das nächste anliegende ASCII-Zeichen nicht mehr als Ziffer interpretierbar ist. Die Adresse addr2 dieses nicht konvertierbaren Zeichens im String verbleibt ebenso wie das doppelt genaue Konvertierungsergebnis +d2 auf dem Stapel.

3.3. Massenspeicheranbindung

3.3.1. Virtuelle Speichertechnik

Forth bietet dem Anwender zur Verwaltung des Massenspeichers ein sehr einfaches, aber leistungsfähiges Konzept an. Der gesamte Massenspeicher (zum Beispiel Band, Diskette oder Festplatte) ist in aufeinanderfolgende Blöcke von jeweils 1024 Byte eingeteilt. Der Arbeit mit den Massenspeicherdaten eines Blockes erfolgt nun aus der Sicht des Forthnutzers nicht über direkte Lese- oder Schreibzugriffe zum Externspeicher, sondern nach Angabe der Nummer des zu bearbeitenden Blockes über normale Speicherzugriffe zu einem reservierten Hauptspeicherbereich, dem sogenannten Block-Puffer-Bereich. Der eigentliche Massen-

speicherzugriff erfolgt verdeckt innerhalb derjenigen Befehle, die Forth zur Umwandlung der Massenspeicheradresse (Blocknummer) in die tatsächliche Hauptspeicheradresse zur Verfügung stellt (virtuelles Speicherprinzip).

3.3.2. Verwalten der Datenblöcke

Das Wort **BLOCK (u => addr)** liefert die Anfangsadresse eines 1024 Byte langen Puffers, in dem der Inhalt des Blocks mit der Blocknummer u steht. BLOCK arbeitet intern folgende Operationen ab: Falls der Block u noch nicht in einem der Puffer verfügbar ist (andernfalls ist keine weitere Bearbeitung erforderlich), erfolgt die Zuweisung eines Puffers. Steht ein als geändert gekennzeichneter Block in diesem Puffer, wird dieser vor der Neuvergabe des Puffers zurückgeschrieben. Dann wird der Block mit der Nummer u vom Massenspeicher in den Puffer transferiert. Die Funktion **BUFFER (u => addr)** ist inhaltlich identisch mit der Funktion BLOCK bis auf den Umstand, daß die Blockleseaktion nicht ausgeführt wird. Der Pufferinhalt ab addr ist damit unbestimmt. Sinnvoll läßt sich dieses Wort anwenden, wenn ein Block vollständig neu beschrieben werden soll; eine vorherige Leseaktion wäre in diesem Fall unnötig. Es gilt die Konvention, daß nur der zuletzt mit BLOCK oder BUFFER angesprochene Pufferinhalt gültig ist. Mit dem Wort **UPDATE** kann dem Forthsystem mitgeteilt werden, daß der Pufferinhalt des aktuell in Bearbeitung befindlichen Blocks aktualisiert worden ist. So markierte Blöcke werden später automatisch auf den Massenspeicher geschrieben, wenn der Puffer für das Speichern eines anderen Blocks benötigt wird. Bei Beenden der Arbeit mit dem Massenspeicher oder auch nur zu Sicherungszwecken ist es sinnvoll, alle als aktualisiert gekennzeichneten Blöcke auf den Massenspeicher zu transferieren, ohne daß vorher Leseaktionen anderer Blöcke erfolgen müssen. Eine solche Funktion erfüllt **SAVE-BUFFERS**. Dieses Wort läßt die Pufferinhalte unverändert und setzt die Aktualisierungskennung zurück. Eine ähnliche Reaktion wird durch das Wort **FLUSH** ausgelöst. Auch hier werden die aktualisierten Blöcke sofort auf den Massenspeicher geschrieben. Im Gegensatz zu **SAVE-BUFFERS** werden aber sämtliche Blockpuffer freigegeben, so daß bei einem neuen Zugriff auf einen Block eine Leseoperation auf dem Massenspeichermedium notwendig wird. Sinnvoll ist diese Funktion zum Beispiel anwendbar, wenn ein Wechsel des Massenspeichermediums vorgesehen ist. So wird vermieden, daß die noch im Puffer befindlichen Blöcke des alten Mediums fälschlicherweise auch als Blöcke des neuen Mediums interpretiert werden. Der Veranschaulichung der Arbeit mit dem Massenspeicher soll ein einfaches Beispiel dienen: Es ist das externe Abspeichern von 16-Bit-Daten zu organisieren:

```
100   CONSTANT  BLOCK0
1024  CONSTANT  B/BUF

:  POSITION
   ( messwertnr ===> pufferoffset blocknr)
   2  *  B/BUF  /MOD  SWAP  BLOCK0  +  ;

:  !WERT        ( wert messwertnr ===> )
   POSITION  BLOCK  +  UPDATE  !  ;
```

Hier markiert **BLOCK0** die Nummer eines Massenspeicherblocks, von dem ab die Aufzeichnung der 16-Bit-Daten erfolgen soll. **B/BUF** ist eine Konstante, die die Länge eines Blocks bzw. eines Blockpuffers angibt. Durch **POSITION** wird unter Benutzung von BLOCK0 und B/BUF das Umrechnen der Datenwertnummer (virtuelle Adresse) in die Massenspeicheradresse (Blocknummer, Position innerhalb des Blocks) vorgenommen. Die Operation **!WERT** zum Abspeichern eines 16-Bit-Datenwertes benutzt wiederum zunächst POSITION, macht den entsprechenden Massenspeicherblock mit BLOCK verfügbar, markiert mit UPDATE den Block als geändert und nimmt dann die eigentliche Änderung durch das Eintragen des Meßwertes auf die errechnete Pufferposition vor. Die ab BLOCK0 abgespeicherten Meßdaten können mit dem nachfolgend definierten Zugriffsbefehl wieder gelesen werden:

```
:  @WERT          ( messwertnr  ===>  wert)
   POSITION  BLOCK  +  @  ;
```

Die Operationen **!WERT** und @**WERT** arbeiten in weitgehender Analogie zu den normalen Speicherzugriffsbefehlen ! und @. Allerdings sind die bei !WERT und @WERT benutzten Meßwertnummern die Adressen eines „echten" 16-Bit-Speichers, das heißt, zwei aufeinanderfolgende Adressen adressieren zwei sich nicht überlappende 16-Bit-Speicherzellen. Nachfolgend ein Bedienungsbeispiel:

```
10 0 'WERT 15 1 !WERT 30 2 !WERT (cr)_ok
0 @WERT . 1 @WERT . 2 @WERT . (cr)_10_15_30_ok
```

Das vorgestellte Massenspeicherkonzept ist allen Systemen gemeinsam, die dem Standard entsprechen. Allerdings lassen sich durch unterschiedliche Implementierungen der Lese- und Schreiboperationen, die in BLOCK und BUFFER benutzt werden, sehr verschiedene Forthsysteme erzeugen, ohne daß dabei eine Verletzung des Standards vorliegen würde. So ist es in Standalone-Versionen üblich, selbstdefinierte physische Gerätetreiber in die Schreib- und Leseoperationen einzusetzen, wobei keine Kompatibilität mit anderen Datenformaten erforderlich ist. Werden dagegen die Befehle BUFFER und BLOCK bei vorhandenem Hostbetriebssystem auf die BIOS-Ebene aufgesetzt, so wird die Kompatibilität des Forthsystems zum Hostbetriebssystem auf der Ebene des physischen Datenformats erreicht. In beiden Fällen stellt sich der gesamte Massenspeicher aus der Sicht des Forthprogrammierers als eine homogene Einheit dar, die nur durch die Blockeinteilung strukturiert ist. Vielfach benutzen Forthsysteme die Lese- und Schreiboperationen eines Hostbetriebssystems auf der DOS-Ebene. Zum einen wird hierdurch eine Dateistrukturierung der Forthprogrammquellen und der Daten möglich; diese brauchen nun nicht mehr nur durch die Blocknummer adressiert werden. Eine Handhabung im klassischen Sinn bleibt weiterhin möglich, indem immer nur über ein- und derselben Datei gearbeitet wird. Zum anderen können die vom Betriebssystem gebotenen Systemhilfen und Standardprogramme genutzt werden. Außerdem kann der Vertrieb von Forthsystemen durch die Orientierung auf Standardbetriebssysteme vereinfacht werden. Systeme mit DOS-Anbindung stellen dem Programmierer üblicherweise Worte zur Dateibehandlung zur Verfügung, die sich weitgehend an den vom Betriebssystem angebotenen Funktionen orientieren.

3.3.3. Speichern von Quelltext

Größere Quellprogramme wird man vor der Kompilation unter Verwendung eines Editors erzeugen und auf dem Massenspeicher konservieren. In Forth wird solcher Quelltext gewöhnlich in Blöcke zu jeweils 1024 Byte gegliedert, wobei ein Block 16 Zeilen mit je 64 Zeichen beinhaltet:

```
   Datei A:FORTH3.CF2   Block 1
 0 \ MP3; Ausgaben                       WD 29-Nov-88
 1
 2 32  CONSTANT  BL
 3
 4 :  SPACE     ( ===> )
 5    BL  EMIT  ;
 6
 7 :  SPACES    ( +n ===> )
 8    0  MAX  ?DUP  IF  0  DO  SPACE
 9                             LOOP
10                      THEN  ;
11
12 :  TREFFER   ( n ===> )
13    ." Das waren "  .   ." Treffer ! "  ;
14
15
```

Für das Erzeugen eines solchen Quellblocks finden sowohl in Forth geschriebene Zeileneditoren als auch komfortable Sreeneditoren Anwendung. Ihre Beschreibung ist im Rahmen des Kurses nicht möglich; Informationen darüber sollten der Dokumentation des entsprechenden Forthsystems entnommen werden.

Ein fertiger Block kann mit dem Befehl **LOAD** interpretiert werden, wobei die Blocknummer auf der Stapelspitze erwartet wird. Dabei gilt für den Blockquelltext die gleiche Syntax, die auch bei der interaktiven Kommandoausgabe gültig ist. So kann ein Quelltext sowohl auszuführende Kommandos als auch neue Definitionen enthalten. Diese Identität bei der Interpretation von Quellblöcken und von interaktiven Eingaben rührt daher, daß durch den Befehl LOAD nur ein zeitweiliges Umlenken des Eingabedatenstroms vorgenommen wird. Die Anzeige des Inhalts eines Quellblocks kann auch ohne einen Editor mit dem Befehl **LIST** erfolgen. Umfaßt der einzugebende Programmquelltext mehrere Blöcke, so können diese durch mehrfache Anwendung der Funktion LOAD geladen werden. Eine andere Möglichkeit besteht darin, Blöcke mit dem Wort --> zu verketten. Das Wort --> steht auf einem Quellblock nach dem letzten Quelltextwort und sorgt dafür, daß auch der nachfolgende Block geladen wird. Das Einfügen von Kommentaren in den Quelltext ist unter Verwendung von runden Klammern, das heißt der Wortkombination (und) möglich. Der Text, der zwischen diesen Zeichen steht, wird vom Interpreter ignoriert. Die öffnende Klammer ist ein normales Forthwort, hinter dem (natürlich) ein Leerzeichen stehen muß! Die schließende Klammer wird analog dem " bei ." als Begrenzerzeichen

verwendet. Ähnlich kann die Kombination .(und) benutzt werden. Sie gibt die zwischen den Klammern stehende Zeichenkette auf dem Terminal aus. Diese Funktion kann zum Beispiel benutzt werden, um beim Laden von langen Quelltextpassagen Zwischeninformationen anzuzeigen.
Forth läßt dem Programmierer viele Freiheiten bei der Formulierung eines Programms. Damit ist die Qualität eines Forthprogramms empfindlich gegenüber der Fähigkeit des Programmierers, diese Freiheiten sinnvoll ausnutzen zu können. Neben solchen Faktoren wie Festlegung der Wortschnittstellen (Dekomposition) und der Namensgebung ist die Gestaltung des Quelltextes von großer Bedeutung für die Übersichtlichkeit, die Lesbarkeit und die Möglichkeiten zur Wartung eines Forthprogramms. An dieser Stelle sollen deshalb einige ausgewählte Empfehlungen für das Layout von Quelltexten genannt werden (siehe auch /2/ in Teil 1):
– Reservieren der Zeile 0 eines jeden Blocks für einen Kommentar, in dem der Zweck des Blocks und der „Stempel" (Initialen des Programmierers und Datum der letzten Revision) enthalten sind
– Benutzen von Leerschritten und Einrükkungen zur Verbesserung der Lesbarkeit
– Notation des Stapeleffekts für jede Doppelpunkt- oder Codedefinition, die Stapelwerte benutzt oder hinterläßt
– Beginn aller Definitionen am linken Zeilenrand, immer nur eine Definition pro Zeile.

wird fortgesetzt

Tafel 3 Kurzbeschreibung der Forthworte

Name	Stapeleffekt	Beschreibung
Konsolfunktionen		
EMIT	(c ===>)	Ausgabe eines Zeichens
KEY	(===> c)	Zeichen einlesen
KEY?	(===> ?)	wahr für Tastatur ist betätigt
CR	(===>)	neue Zeile
SPACE	(===>)	Ausgabe eines Leerzeichens
SPACES	(+n ===>)	Ausgabe von +n Leerzeichen
Zeichenkettenbefehle		
COUNT	(addr1 ===> addr2 +n)	Lesen der Länge, Berechnen des Anfangs
TYPE	(addr +n ===>)	Ausgabe von +n Zeichen ab addr
–TRAILING	(addr +n1 ===> addr +n2)	Abschneiden von Leerzeichen am Ende
EXPECT	(addr +n ===>)	ab addr +n Zeichen einlesen
SPAN	(===> addr)	Anzahl der gelesenen Zeichen in addr
."	(===>)	Ausgabe der Zeichenkette bis "
Konvertierungsbefehle		
<#	(===>)	Bereitstellen einer leeren Kette
#	(+d1 ===> +d2)	Konvertieren einer Stelle
#S	(+d ===> 0 0)	Ausführen von # bis Rest = 0
#>	(32b ===> addr +n)	Abschluß der Konvertierung, Bereitstellen der Ausgabekette
HOLD	(c ===>)	c in Kette einfügen
SIGN	(n ===>)	Vorzeichen von n in Kette einfügen
CONVERT	(+d1 addr1 ===> +d2 addr2)	Konvertieren ab addr1 bis addr2
BASE	(===> addr)	Variable für aktuelle Basis
DECIMAL	(===>)	Zahlenbasis dezimal setzen
HEX	(===>)	Zahlenbasis hexadezimal setzen
weitere Zahlenausgaben		
.R	(n +n ===>)	Anzeige n +n-stellig rechtsbündig
D.R	(d +n ===>)	Anzeige d +n-stellig rechtsbündig
Massenspeicheroperationen		
BLOCK	(u ===> addr)	Lesen von Block u auf addr
BUFFER	(u ===> addr)	Reservieren addr für Block
UPDATE	(===>)	markiert aktiven Puffer als geändert
FLUSH	(===>)	Retten und Leeren aller Puffer
SAVE-BUFFERS	(===>)	Retten aller Puffer
Behandlung des Quelltextstroms		
LIST	(u ===>)	Anzeige Block u
LOAD	(u ===>)	Block interpretieren
-->	(===>)	Weiterladen ab nächstem Block
.(	(===>)	Kommentarausgabe bis)
(<Leerzeichen>	(===>)	Kommentar bis)

Wegbereiter der Informatik



PAFNUTI LWOWITSCH TSCHEBYSCHEW

*** 1821 Okatowo,**
† 1894 St. Petersburg

Der russische Mathematiker P. L. Tschebyschew ist der Gründer der Petersburger Mathematikerschule, aus der zum Beispiel A. A. Markow und A. M. Ljapunow als seine Schüler hervorgegangen sind. Er arbeitete auf verschiedenen Gebieten der Analysis, der Wahrscheinlichkeitstheorie, der Zahlentheorie und der Mechanik.
Tschebyschew stammt aus einer adligen Familie und erhielt seinen Schulunterricht im Elternhaus. 1832 verzog die Familie nach Moskau, um den Söhnen den Besuch der Universität zu erleichtern. Schon 1837 – also mit 16 Jahren – ließ sich Tschebyschew an der physikalisch-mathematischen Fakultät der Moskauer Universität immatrikulieren und beendete sein Studium 1841. Fünf Jahre später erlangte er hier die Magisterwürde und siedelte 1847 nach St. Petersburg über, wo er an der Universität Vorlesungen zur Algebra und zur Zahlentheorie hielt. Im Jahre 1849 verteidigte er seine Dissertation und wurde 1850 Professor an der Petersburger Universität. Dieses Amt bekleidete er bis 1882, danach widmete er sich an der Petersburger Akademie (deren ordentliches Mitglied er 1859 geworden war) ausschließlich seiner wissenschaftlichen Tätigkeit. Nebenher war er Mitarbeiter in verschiedenen Kommissionen bei mehreren russischen Ministerien.
In der mathematischen Wissenschaft, besonders in der Angewandten Mathematik, hat sich Tschebyschew bleibende Verdienste erworben. Bekannt sind die nach ihm benannten Polynome
$T_n(x) = \cos(n \arccos x)$,
die eine besondere Rolle bei der Approximation von Funktionen spielen und einen Sonderfall der Jacobischen Polynome darstellen. Für die numerische Berechnung des bestimmten Integrals einer Funktion fand er die Quadraturformel

$$\int_{-1}^{1} f(x)\,dx \approx \frac{2}{n}\sum_{i=1}^{n} f(x_i),$$

die für Polynome (n – 1)-ten Grades exakt ist (wenn n die Anzahl der Interpolationsknoten bedeutet).
Weniger bekannt geworden ist die Tatsache, daß Tschebyschew auch eine Rechenmaschine konstruiert hat, von ihm Arithmometer (Zahlenmesser) genannt. Sie ist für additive und subtraktive Verarbeitung 10stelliger Zahlen ausgelegt; in ihr ist das Prinzip des gleitenden Zehnerübertrags verwirklicht. Tschebyschew hat das erste Exemplar dieser Maschine dem Pariser Museum für Kunst und Handwerk übergeben und an der dortigen Akademie darüber auch vorgetragen. Ein zweites, verbessertes Exemplar blieb in Petersburg. In der Mechanik hat Tschebyschew eine Theorie der Gelenkmechanismen entwikkelt. Und so entsprang sein Arithmometer wohl hauptsächlich seinem konstruktiven Interesse, den gleitenden Zehnerübertrag zu erproben. Denn die Maschine selbst ist sehr unhandlich und nur mit großem Kraftaufwand zu bedienen, auch fehlt ihr eine Vorrichtung zur Ziffern-Rückstellung oder eine Bedienungskurbel. Es gibt keinerlei Hinweise darauf, daß diese Maschine überhaupt (und sei es von Tschebyschew selbst!) jemals benutzt worden ist.
Tschebyschew, familiär ungebunden, hat zahlreiche Reisen ins Ausland unternommen, sei es, um große Industrieanlagen zu besichtigen oder mit bedeutenden Fachkollegen (Ch. Hermite, J. Bertrand, L. Kronecker) zusammenzutreffen. Er war gewähltes Mitglied der Akademien in Berlin, Bologna, Paris und London.

Dr. Klaus Biener

Einführung in Forth-83

Teil 4

Dr. Hartmut Pfüller (Leiter),
Dr. Wolfgang Drewelow, Dr. Bernhard Lampe,
Ralf Neuthe, Egmont Woitzel
Wilhelm-Pieck-Universität Rostock,
Sektion Technische Elektronik

4. Die virtuelle Forthmaschine

Diese Folge bietet in aller Kürze eine Darstellung der inneren Funktion eines Forthsystems. Während der Durchschnittsprogrammierer diese Informationen im allgemeinen entbehren kann, wird dem interessierten Leser gezeigt, wie die Leistungsfähigkeit von Forth auf wenigen, überraschend einfachen Konzepten beruht. In den vorangegangenen Folgen stand die Beschreibung derjenigen Komponenten von Forth im Mittelpunkt, die die für den Programmierer sichtbare *Oberfläche* des Systems bilden. Bevor in den weiteren Folgen Möglichkeiten zur *Erweiterung* des Systems vorgestellt werden, versucht dieser Beitrag, die *interne Arbeitsweise* eines Forthsystems verständlich zu machen. Im ersten Abschnitt wird der Aufbau des Wörterbuchs behandelt. Die Erläuterung der internen Struktur von Worten macht die Methode transparent, nach der Programme auch über indirekte Bezugnahmen aufgerufen werden können. Im zweiten Abschnitt wird die interne Arbeitsweise von Forth vorgestellt. Auf dieser Grundlage wird dann als drittes die häufig notwendige Einbindung von Maschinencode eingeführt.

4.1. Die interne Wortstruktur

4.1.1. Komponenten eines Wortes

Die fundamentalen Bausteine eines Forthsystems sind die Worte. Sie müssen unter zwei Aspekten betrachtet werden. Einerseits sind Worte ausführbare Softwaremoduln, wobei durch ihre Struktur verborgen wird, ob es sich um Datenmoduln (z. B. Variablen oder Konstanten) oder Programmoduln (Doppelpunktdefinition) handelt. Andererseits stellen die Worte auch über ihre Namen die Verbindung zum Bediener her. Entsprechend dieser funktionellen Zweiteilung besteht jeder Eintrag im Wörterbuch aus zwei gekoppelten Teilen, die *Kopf* und *Rumpf* genannt werden. Der Kopf enthält dabei alle Informationen, die für die Dialogarbeit erforderlich sind. Dazu gehören der Name des Wortes, der im dafür vorgesehenen *Namenfeld* abgespeichert wird, sowie die Einordnung des Wortes in die Suchreihenfolge, die durch das sogenannte *Verkettungsfeld* geschieht. Der Rumpf nimmt dagegen alle diejenigen Informationen auf, die für die Abarbeitung des Wortes erforderlich sind. Im einzelnen sind dies im *Parameterfeld* liegende *Parameter* des Wortes und – ganz wichtig (!) – der im sogenannten *Codefeld* gespeicherte Verweis auf die bei Aufruf des Wortes auszuführende Aktion (Bild 4.1). Durch den Standard Forth-83 wird die konkrete Anordnung der einzelnen Felder nicht festgelegt; als Beispiel für die Erläuterungen hier dient das System comFORTH 2. Da die Größe und die Reihenfolge der einzelnen

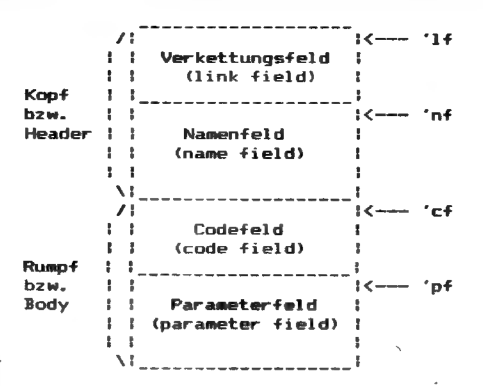


Bild 4.1 Aufbau eines einzelnen Wörterbucheintrags; 'lf bedeutet Linkfeldadresse, 'nf Namenfeldadresse usw.

Wortkomponenten in den Systemen verschiedener Hersteller oft unterschiedlich sind, bietet der Standard in einem experimentellen Vorschlag einen Wortschatz an, der wenigstens die Umrechnung der verschiedenen Anfangsadressen ineinander gestattet. Als einheitlicher Bezugspunkt wird dabei die Codefeldadresse 'cf benutzt. Aus ihr können mit Hilfe der Operationen **>LINK**, **>NAME** und **>BODY** die Link-, Namens- bzw. Parameterfeldadresse berechnet werden. Die entsprechenden Rückrechnungen erledigen die Worte **LINK>**, **NAME>** und **BODY>**. An sich stellen diese sechs Operationen bereits einen vollständigen Funktionensatz bereit, da auf dem Umweg über die Codefeldadresse aus jeder Adresse eine beliebige andere berechnet werden kann. Es ist jedoch so, daß aufgrund der Funktionsteilung zwischen Kopf und Rumpf beide Bestandteile nicht unbedingt in demselben Speicherbereich aufbewahrt werden müssen. So kann unter Umständen die physische Trennung von Kopf und Rumpf Vorteile bieten, zum Beispiel die Entlastung des 16-Bit-Adreßraums der Forthmaschine. Falls mit einer solchen Variante gearbeitet wird, ist für die Aufrechterhaltung der Systemfunktion nur ein Adreßverweis vom Kopf zum Rumpf zwingend erforderlich. Da in diesem Fall die beiden Funktionen >LINK und >NAME nicht mehr oder nur mit unverhältnismäßig hohem Aufwand implementiert werden können, ermöglichen die beiden Worte **N>LINK** und **L>NAME** die direkte Umrechnung von Namenfeld- und Verkettungsfeldadresse.

4.1.2. Suchoperationen

Suchoperationen über der Datenstruktur Wörterbuch sind an verschiedenen Stellen innerhalb des Systems erforderlich. Bei der Analyse von Quelltext muß z. B. der Textinterpreter untersuchen, ob die eingegebene Zeichenkette ein Wort ist, also ob ein entsprechender Eintrag in der aktuellen Suchordnung existiert. Für die Ausführung der Suchoperationen werden die im Kopf gespeicherten Informationen benutzt. Der Suchablauf im Wörterbuch wird zuerst durch die im Teil 2 des Kurses (siehe MP 5/1989) diskutierte Reihenfolge der zu durchsuchenden Vokabulare bestimmt. Innerhalb der Vokabulare legen die Verkettungsfelder die Suchreihenfolge fest. Die Suche erfolgt dabei umgekehrt zur zeitlichen Definitionsreihenfolge, das zuletzt definierte Wort wird also zuerst gefunden, danach das vorletzte usw. So wird auf einfache Weise erreicht, daß bei mehreren Definitionen gleichen Namens immer die jüngste als gültig gefunden wird. Erreicht wird dies durch eine Technik, die als verkettete Liste bezeichnet wird. Bild 4.2 zeigt den prinzipiellen Aufbau einer solchen Datenstruktur. Das Verkettungsfeld im Kopf eines Forthwortes enthält dementsprechend einen Adreßverweis zum Kopf des direkt zuvor in derselben Liste definierten Wortes. Im System comFORTH 2 ist an dieser Stelle die Linkfeldadresse des Vorgängers zu finden. Der Standard schreibt dies jedoch nicht vor, und es gibt auch Systeme, die jeweils auf die Namenfeldadresse des Vorgängers verweisen.

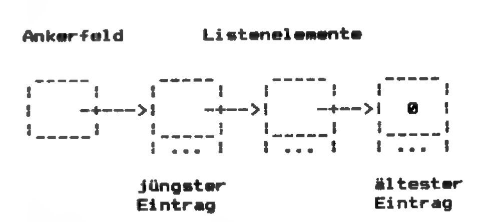


Bild 4.2 Verkettete Liste

Zum Vergleich der zu suchenden Zeichenkette mit dem Namen des Wortes dient das Namenfeld. Auch dessen Aufbau ist nicht standardisiert. Während comFORTH 2 den Namen in voller Länge abspeichert (siehe Bild 4.3), werden durch polyFORTH z. B. nur die ersten drei Zeichen abgespeichert, um Speicherplatz zu sparen und die Suchgeschwindigkeit zu erhöhen.

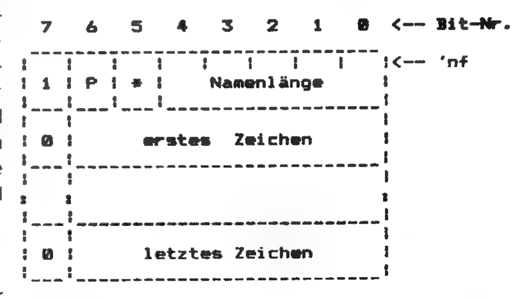
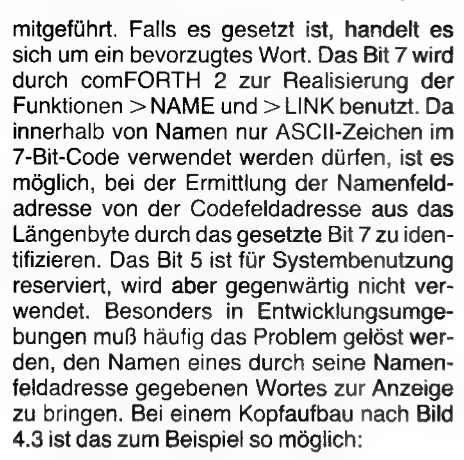


Bild 4.3 Namenfeldaufbau im System comFORTH2

Im Namenfeld sind zusätzlich zum Wortnamen einige weitere Informationen gespeichert. Dazu gehört insbesondere die Information, ob es sich um ein sogenanntes *bevorzugtes Wort* handelt, das unabhängig vom Systemzustand ausgeführt werden soll (immediate word). Die Diskussion dieser Worte ist Gegenstand der Folge 5. In comFORTH 2 wird diese Information im P-Bit auf Position 6 des Längenbytes im Namenfeld



mitgeführt. Falls es gesetzt ist, handelt es sich um ein bevorzugtes Wort. Das Bit 7 wird durch comFORTH 2 zur Realisierung der Funktionen >NAME und >LINK benutzt. Da innerhalb von Namen nur ASCII-Zeichen im 7-Bit-Code verwendet werden dürfen, ist es möglich, bei der Ermittlung der Namenfeldadresse von der Codefeldadresse aus das Längenbyte durch das gesetzte Bit 7 zu identifizieren. Das Bit 5 ist für Systembenutzung reserviert, wird aber gegenwärtig nicht verwendet. Besonders in Entwicklungsumgebungen muß häufig das Problem gelöst werden, den Namen eines durch seine Namenfeldadresse gegebenen Wortes zur Anzeige zu bringen. Bei einem Kopfaufbau nach Bild 4.3 ist das zum Beispiel so möglich:

```
: .NAME      ( 'nf ===> )
  COUNT  31  AND     TYPE     ;
```

Da der Name in voller Länge gespeichert wird, ist nach Maskierung der höherwertigen Bits des Längenbytes (mittels 31 = 1 FH) direkt das Wort **TYPE** anwendbar. Die Kenntnis der genannten Implementierungsdetails ist bei Anwendung der nachfolgend beschriebenen, standardisierten Suchworte nicht erforderlich. Ein für die Dialogarbeit sehr wichtiges Wort ist das Häkchen (der Apostroph) mit dem Zeichen ' (Aussprache: *tick*). Es ermittelt die Codefeldadresse des im Eingabestrom folgenden Wortes. Als Beispiel dient folgende Kommandozeile:

' DROP >NAME .NAME <u>(cr) DROP ok</u>

Das Häkchen ermittelt hier die Codefeldadresse des Wortes **DROP**. Durch >NAME wird sie in die Namenfeldadresse gewandelt und durch das oben definierte **.NAME** der Name angezeigt. Das Häkchen besitzt bei seiner Verwendung in Anwenderdefinitionen einen Nachteil. Falls die gegebene Zeichenkette nicht in der aktuellen Suchordnung gefunden werden kann, bricht die Suche mit der Fehlermeldung *unbekannt* das aufrufende Wort oder die bearbeitete Kommandozeile mit einer Fehlermitteilung ab. Falls dies nicht erwünscht ist, muß das elementare Suchwort **FIND** verwendet werden, das in der aktuellen Wörterbuchordnung nach einem Wortnamen sucht. FIND liefert als Ergebnis zusätzlich zur Codefeldadresse einen als Flag verwendbaren Parameter. Dieser ist ungleich Null, falls das Wort gefunden wurde, gleich −1, falls es sich um ein einfaches oder gleich 1, falls es sich um ein bevorzugtes Wort handelt.

4.1.3. Indirekter Wortaufruf

Im interpretierenden Systemzustand veranlaßt der Textinterpreter sofort die Abarbeitung eines jeden Wortes, das er mit Hilfe von FIND in der Suchordnung gefunden hat. Dazu benutzt er das Wort **EXECUTE**, dem als Eingangsparameter die Codefeldadresse des auszuführenden Wortes übermittelt wird. EXECUTE und ' sind in ihrem Verhalten aufeinander abgestimmt. Im folgenden Beispiel wird – zu Demonstrationszwecken über den Umweg ' und EXECUTE – das Wort . ausgeführt:

123 ' . EXECUTE <u>(cr) 123 ok</u>

Die Möglichkeit zum Aufruf von Worten über EXECUTE wird gern benutzt, um das Verhalten von Programmen auch nach ihrer Übersetzung noch modifizieren zu können. Im folgenden kleinen Beispiel soll diese als indirekte oder vektorisierte Ausführung bekannte Technik vorgestellt werden. Angenommen, in einem Programm soll es möglich sein, den Stil der Fehlerausschriften zu wechseln. Dazu kann eine

VARIABLE FEHLERTEXT

definiert werden, die die Codefeldadresse desjenigen Wortes enthalten soll, das den Text ausgibt. Zur Auswahl stehen die Worte

```
:  ERNST      ."Fehler"    ;
:  LUSTIG     ."Pech gehabt"    ;
```

Mit Hilfe des Häkchens kann der Zeigervariablen die Codefeldadresse zum Beispiel des „ernsten" Textes zugewiesen werden:

' ERNST FEHLERTEXT !

Zur konkreten Meldungsausgabe wird ein Wort definiert, das die Codefeldadresse aus der Variablen ausliest und mit Hilfe von EXECUTE den aktuellen Text zur Ausgabe bringt:

: MELDUNG FEHLERTEXT @ EXECUTE ;

Zunächst bringt der Aufruf von MELDUNG den oben eingetragenen „ernsten" Text:

MELDUNG <u>(cr) Fehler ok</u>

Durch Einspeichern der Codefeldadresse für den „lustigen" Text kann das Verhalten von **MELDUNG** zu jedem Zeitpunkt verändert werden. Es ist auch möglich, zum Eintragen der entsprechenden Codefeldadressen spezielle Worte zu definieren.

```
:  LUSTIGE  [']  LUSTIG  FEHLERTEXT  !  ;
:  ERNSTE   [']  ERNST   FEHLERTEXT  !  ;
```

Hier wird mit dem Wort **[']** ein Verwandter des Häkchens benutzt. Dieses Wort arbeitet genauso wie ', nur daß es bereits zur Übersetzungszeit ausgeführt wird. Die Kommandofolge **[']** **LUSTIG** führt bei Ausführung des Wortes LUSTIGE zur Übergabe der Codefeldadresse von LUSTIG auf dem Parameterstapel. Die beiden oben definierten Worte können folgendermaßen benutzt werden:

LUSTIGE MELDUNG <u>(cr) Pech gehabt ok</u>
ERNSTE MELDUNG <u>(cr) Fehler ok</u>

4.2. Der virtuelle Forthprozessor

Während im Abschnitt 4.1. die Dialogarbeit im Vordergrund stand, wird im weiteren die interne Arbeitsweise eines Forthsystems beschrieben. Hier liegt auch der Schlüssel für die hohe Portabilität der Forthumgebung.

In der Schichtarchitektur eines Forthsystems befindet sich unmittelbar über der Hardware eine Schicht, die virtuelle Maschine genannt wird. In der Tat modelliert diese, im Code des Wirtsrechners programmierte Schicht eine sehr einfache Rechnerkonfiguration. Bild 4.4 gibt einen Überblick über deren Aufbau.

Das Kernstück dieses Rechners ist die CPU, der virtuelle Forthprozessor. Diesem stehen mehrere unabhängige Speicherbereiche zur Verfügung. Da sind der Parameterstapel als Arbeitsspeicher, ein Rückkehrstapel für Organisationszwecke sowie der Hauptspeicher zur Aufnahme von Programmen und Daten. Der Befehlssatz dieses Prozessors ist nicht starr begrenzt, sondern kann erweitert werden. Das erfolgt in der Maschinensprache der verwendeten Hardware. Dieses Verfahren ist am ehesten mit der Mikroprogrammierung vergleichbar. Aus der Sicht des Forthsystems ist es deshalb zweckmäßig, den Maschinencode des Wirtsrechners als *Mikrocode* zu bezeichnen. Der Forthprozessor besitzt einen sehr eingeschränkten Registersatz (siehe Bild 4.5). Er benötigt neben den drei für die Ablaufsteuerung erforderlichen Registern *MIP*, *IP* und *WA* nur zwei Zeigerregister *PSP* und *RSP* für die beiden Stapelspeicher. Die herkömmlichen Allzweckregister werden nicht benötigt, da fast alle Befehle ausschließlich mit Stapeladressierung arbeiten.

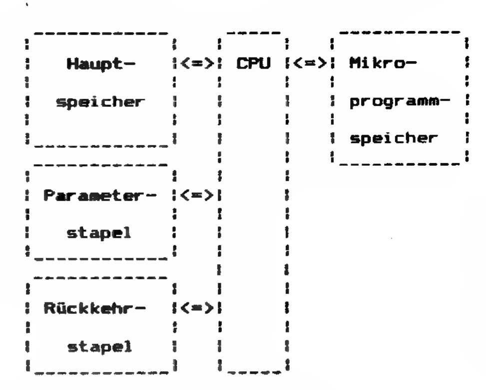


Bild 4.4 Virtuelle Forthmaschine

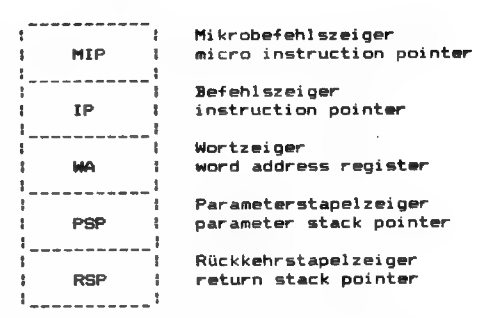


Bild 4.5 Registersatz des virtuellen Forthprozessors

4.2.1. Der Fadencodeaufbau

Der Forthprozessor besitzt eine eigene Maschinensprache, die auf eine Technik zurückzuführen ist, die als Fadencode bekannt wurde. Fadencode wird durch den Forthübersetzer während der Kompilation von Doppelpunktdefinitionen erzeugt. Für die Implementierung von Fadencode gibt es verschiedene, qualitativ jedoch gleichwertige Varianten. Am häufigsten wird die doppelt indirekte Technik verwendet, auf die sich die weiteren Aussagen beziehen.

Bei dieser Fadencodevariante wird die letztlich aufzurufende Mikrocodepassage in zwei Stufen indirekt verschlüsselt. Die erste Stufe ist der Fadencode selbst, der aus einer Aufzählung der Codefeldadressen der nacheinander aufzurufenden Worte besteht. Die zweite Verschlüsselung erfolgt auf den Codefeldern der Worte. Diese enthalten die Adresse des Mikroprogramms, das bei Aufruf des jeweiligen Wortes auszuführen ist. Diese Mikroprogramme organisieren die Behandlung der Parameterfelder, deren Aufbau von der Art des Wortes abhängt. Auf diese Weise

können alle Wortarten mit derselben Methode aufgerufen werden. Bild 4.6 zeigt den Aufbau des Fadencodes, der bei einer Definition

: TUCK SWAP OVER ;

erzeugt würde. Hier wird angenommen, daß sowohl **SWAP** als auch **OVER** Mikroprogramme sind. Deshalb verweisen deren Codefelder jeweils auf die eigenen Parameterfelder, die den zugehörigen Mikrocode enthalten. Solche Worte werden auch Primärworte genannt.

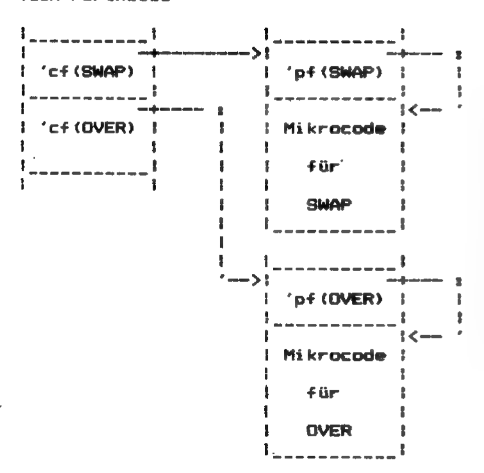


Bild 4.6 Beispiel für Forthcode

4.2.2. Der Adreßinterpreter

Die Abarbeitung von Fadencode erfolgt durch die Ablaufsteuerung des virtuellen Forthprozessors, die aufgrund ihrer Arbeitsweise oft Adreßinterpreter genannt wird. Da sie die Ausführung der jeweils nächsten Anweisung organisiert, besitzt die Einsprungstelle meist den Namen **NEXT**. Diese Ablaufsteuerung benötigt zu ihrer Arbeit drei Register (siehe Bild 4.5). Die Funktion des Adreßinterpreters kann in der Sprache eines fiktiven Prozessors folgendermaßen beschrieben werden:

```
NEXT: LD   WA,(IP)   ;WA zeigt auf 'cf
      INC  IP        ;IP weiter setzen
      LD   MIP,(WA)  ;Mikroprogramm aus
                     ;Codefeld anspringen
```

Zentrale Bedeutung für die Ablaufsteuerung hat der Befehlszeiger **IP**. Beim Eintritt in NEXT muß er auf die nächste zu bearbeitende Fadencodeanweisung zeigen. Dann wird zuerst die Codefeldadresse des aufzurufenden Worts in das *WA*-Register gebracht und der Befehlszeiger auf die folgende Fadencodeanweisung vorgerückt. Anschließend wird der Mikrobefehlszähler (also der Befehlszähler der Wirtsmaschine) mit derjenigen Adresse geladen, die auf dem Codefeld des abzuarbeitenden Wortes steht. Damit wird die Steuerung an das entsprechende Mikroprogramm abgegeben, das seinerseits unbedingt mit einem Sprung zurück zu NEXT enden muß. Bild 4.7 zeigt eine typische Implementierung für den Prozessor U 880.

```
; --------- Registerbelegung ----------

; Befehlszeiger                - BC
; Wortzeiger                   - DE
; Parameterstapelzeiger        - SP
; Rückkehrstapelzeiger         - Speicherzelle
; Mikroprogrammzeiger          - PC

NEXT: LD    A,(BC)    ;lesen low('cf)
      INC   BC        ;IP ein Byte weiter
      LD    L,A       ;HL ist WA
      LD    A,(BC)    ;high('cf) analog
      INC   BC        ;nächster Befehl
      LD    H,A       ;'cf vollständig
      LD    E,(HL)    ;DE ist Zwischen-
      INC   HL        ;speicher für MIP
      LD    D,(HL)    ;WA:='cf+1 !!!
      EX    DE,HL     ;DE wird WA
      JP    (HL)      ;laden MP
```

Bild 4.7 Ablaufsteuerung für den Prozessor U 880

Der Fadencode besteht nicht ausschließlich aus einer Aneinanderreihung von Codefeldadressen. Verschiedenen Worten werden auch mit Hilfe der unmittelbaren Adressierung Parameter übermittelt. Zu diesen Worten gehören zum Beispiel die im Systemerweiterungswortschatz standardisierten Worte **BRANCH** und **?BRANCH**, mit denen bedingte und unbedingte Sprünge innerhalb des Fadencodes realisiert werden. Bei ihnen findet man unmittelbar im Fadencode folgend das Sprungziel (Bild 4.8). In der Sprache des fiktiven Prozessors formuliert ruft BRANCH das folgende Mikroprogramm auf:

```
BRANCH: LD  IP,(IP)    ;laden Zieladresse
        LD  MIP,NEXT   ;Rückkehr zur
                       ;Ablaufsteuerung
```

BRANCH und ?BRANCH werden beispielsweise durch die Worte **IF** und **ELSE** benutzt. Einzelheiten dazu sind Gegenstand der nächsten Folge.

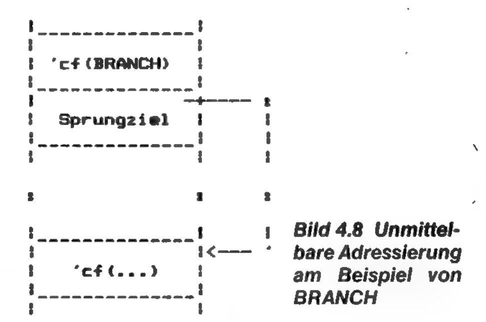


Bild 4.8 Unmittelbare Adressierung am Beispiel von BRANCH

4.2.3. Aufruf von Fadencodeprogrammen

Eines der wesentlichen Merkmale von Fadencode ist, daß die Art des Wortes durch den Inhalt seines Codefeldes verschlüsselt ist. Damit kann der Aufruf von Primärworten genauso erfolgen, wie der von Doppelpunktworten (Sekundärworten). Dafür verantwortlich ist das durch diese Worte aufgerufene Mikroprogramm mit dem Namen **DOCOL** (von englisch *do-colon*). Es entspricht der CALL-Anweisung herkömmlicher Prozessoren:

```
DOCOL: DEC RSP       ;Platz auf RS
       LD  (RSP),IP  ;retten IP
       INC WA        ;WA: = 'pf
       LD  IP,(WA)   ;IP auf Programm
       LD  MIP,NEXT  ;Ablaufsteuerung
```

Durch DOCOL wird der momentane Stand des Befehlszeigers *IP*, also die Fortsetzungsadresse, auf den Rückkehrstapel gerettet.

Da sich das aufzurufende Fadencodeprogramm auf dem Parameterfeld des gerufenen Wortes befindet, wird dessen Adresse in den Befehlszeiger eingetragen. Die Rückkehr aus einem so aufgerufenen Fadencodeprogramm wird durch das am Ende jedes Fadencodeprogramms kompilierte Primärwort **EXIT** organisiert.

```
EXIT: LD  IP,(RSP)  ;Fortsetzungsadresse
      INC RSP       ;Platz freigeben
      LD  MIP,NEXT  ;Ablaufsteuerung
```

EXIT entspricht der Return-Anweisung üblicher Prozessoren. Das Zusammenspiel von DOCOL und EXIT soll anhand des Aufrufs des oben definierten Wortes **TUCK** aus dem Wort

```
: 2!     ( ps: 32b addr ===> )
  TUCK   ! 2+ ! ;
```

illustriert werden. Bild 4.9 zeigt den Aufbau des zugehörigen Fadencodes. Die Adresse X und Y bezeichnen die Parameterfeldadresse von 2! bzw. TUCK.

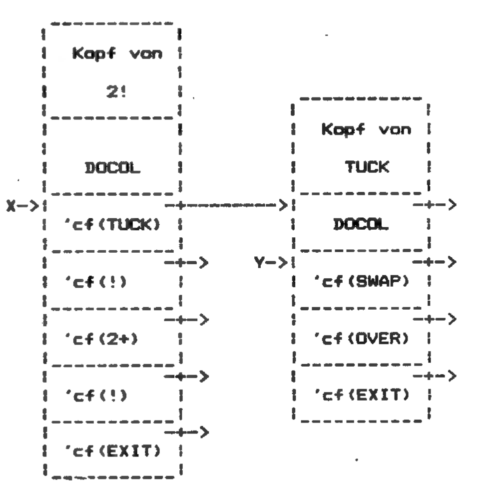


Bild 4.9 Interne Repräsentation der Worte TUCK und 2!

In der Tafel 4.1 kann die Registerbelegung und die Stapelbelegung während der Abarbeitung von **2!** verfolgt werden. Als Ausgangssituation wird der Aufruf von 2! aus einem Fadencodeprogramm auf der Adresse *ip* angenommen. In der ersten Spalte sind die durch die Ablaufsteuerung nacheinander aufgerufenen Mikroprogramme, in den weiteren der Zustand nach ihrer Ausführung aufgeführt. Die als *pn* bzw. *rn* aufgeführten Parameter werden nicht beeinflußt.

Tafel 4.1 Abarbeitung des Wortes 2!

Mikro-programm	Parameterstapel				Rückkehrstapel		IP
	3	2	1	0	1	0	
...	p3	1(32b)	h(32b)	addr	r1	r0	ip
DOCOL	p3	1(32b)	h(32b)	addr	r0	ip+2	X
DOCOL	p3	1(32b)	h(32b)	addr	ip+2	X+2	Y
SWAP	p3	1(32b)	addr	h(33b)	ip+2	X+2	Y+2
OVER	1(32b)	addr	h(32b)	addr	ip+2	X+2	Y+4
EXIT	1(32b)	addr	h(32b)	addr	r0	ip+2	X+2
!	p4	p3	1(32b)	addr	r0	ip+2	X+4
2+	p4	p3	1(32b)	addr+2	r0	ip+2	X+6
!	p6	p5	p4	p3	r0	ip+2	X+8
EXIT	p6	p5	p4	p3	rl	r0	ip+2

4.2.4. Der Rückkehrstapel

Der Rückkehrstapel darf mit Vorsicht auch für das zwischenzeitliche Speichern von Daten verwendet werden. Zu diesem Zweck bietet der Standard drei Funktionen an, die den Transfer von Daten zwischen dem Parameterstapel und dem Rückkehrstapel ermöglichen. Dies sind:

```
>R  ps:  16b ===> rs: ===> 16b
R>  ps:  ===> 16b rs: 16b ===>
R@  ps:  ===> 16b rs: 16b ===> 16b
```

Bei der Benutzung dieser Befehle ist zu beachten, daß der Rückkehrstapel durch die Ablaufsteuerung benutzt wird. Sie sollten deshalb nur innerhalb einer Aufrufebene verwendet werden. Es ist darauf zu achten, daß vor dem Rücksprung in die aufrufende Programmebene alle Parameter wieder entfernt werden. Diese Vorgehensweise entspricht der Benutzung des Prozessorstapels bei Assemblerprogrammierung. Trotz dieser Einschränkungen bleibt der Rückkehrstapel ein wichtiges Hilfsmittel zur Vermeidung umfangreicher Stapelmanipulationen. Als Beispiel dafür kann die Hochwortdefinition des Wortes **2SWAP** herangezogen werden.

```
: 2SWAP ( ps: 32b ===> 32b )
  ROT >R ROT R> ;
```

Ohne Verwendung des Rückkehrstapels hätte man die wesentlich langsamere Variante

```
: 2SWAP  3 ROLL  3 ROLL  ;
```

verwenden müssen.

Mit Hilfe des Rückkehrstapels ist es auch möglich, Fadencodeprogramme indirekt anzuspringen und auszuführen. Dazu braucht nur deren Adresse auf den Rückkehrstapel gelegt werden. Beim nächsten Aufruf von EXIT wird diese dann in das *IP*-Register geladen und das zugehörige Programm bearbeitet. (Dieses Verfahren ist auch in der Assemblerprogrammierung bekannt: Eine Zieladresse wird dadurch angesprungen, daß sie auf den Stapel gelegt wird und daß danach ein Returnbefehl ausgeführt wird.) Die (nicht interaktiv ausführbare) Kommandofolge

```
'. >BODY  >R
```

führt in diesem Sinne zum Aufrufen des Fadencodes der Zahlenausgabe. Diese Methode ist auf Fadencode beschränkt und benötigt nicht die Existenz eines Codefeldes.

4.3. Mikroprogrammierung

Bei der Programmierung prozeßnaher Anwendungen taucht immer wieder das Problem auf, Teile des Gesamtprogramms im Maschinencode des Hardwareprozessors formulieren zu müssen. Dies betrifft insbesondere sehr zeitkritische Abschnitte oder die Bedienung spezieller Peripherie. In einem Forthsystem ist solcher Code sehr organisch in Form neuer Mikroprogramme des Forthprozessors integrierbar.

4.3.1. Assembler in Forth

Für das Erzeugen von Primärworten wird durch den Standard im Erweiterungswortschatz für Assemblerprogrammierung die Konstruktion

```
CODE  name  ...  END-CODE
```

definiert. Ihre Benutzung erfolgt analog zur Konstruktion

```
:  name  ...  ;
```

zur Definition von Fadencodeprogrammen. Anstelle der Aufzählung von Forthworten erfolgt die Angabe der Assemblerbefehle, die nacheinander ausgeführt werden sollen. Im Gegensatz zur Doppelpunktkonstruktion verbleibt das System zwischen **CODE** und **END-CODE** im Ausführmodus.

Die unter Regie des Forthsystems benutzten Assemblerbefehle sind wie alle anderen Kommandos ebenfalls in Form von Forthworten verfügbar. Sie werden in dem Vokabular **ASSEMBLER** aufbewahrt, das durch CODE automatisch in die Wörterbuch-Suchordnung aufgenommen und durch END-CODE wieder aus dieser entfernt wird.

Die Worte des Assemblervokabulars unterliegen keiner Standardisierung. Es haben sich jedoch einige Konventionen durchgesetzt, die im weiteren beschrieben werden. Entsprechend der Systemphilosophie benutzt dieser Assembler die umgekehrt polnische Notation. Das bedeutet, daß zuerst die Operanden, zum Beispiel Registernamen, und danach die Operationen angegeben werden. Bei Zweioperandenbefehlen wird analog zum Wort ! zuerst der Quelloperand und danach der Zieloperand erwartet. Zur Unterscheidung der Mnemoniks von standardisierten Worten wie **AND** oder **OR** enden alle Assemblerworte mit einem Komma. Da das System im Ausführmodus verbleibt, können Zahlen als Direktoperanden einfach über den Stapel übergeben werden. Dabei ist es belanglos, ob sie direkt als Zahl eingegeben werden oder das Ergebnis einer Berechnung sind. Zur Sicherung der Syntax werden sie mit dem Wort # markiert, das im Assemblervokabular eine spezielle Bedeutung besitzt. Die Tafel 4.2 zeigt dies am Beispiel des Additonsbefehls des U 880.

Tafel 4.2 Forth-Assemblernotation

symbolische Notation	Original-Notation	FORTH-Notation
ADD r	ADD L	L ADD,
ADD n	ADD 12	12 # ADD,
ADD ii+d	ADD (IX+7)	7 (IX) ADD,
ADD HL,dd	ADD HL,DE	DE HL ADD,

4.3.2. Programmierbeispiel

Als Abschluß dieses Abschnitts soll anhand der Implementierung des oben mit Hilfe des Doppelpunkts definierten Wortes TUCK ein Beispiel für den Umgang mit dem U 880-Assembler vorgeführt werden.

Der in Bild 4.10 zu findende Quelltext bezieht sich auf die in Bild 4.7 gezeigte Ablaufsteuerung für den U 880. Diese verwendet das Register *BC* als Befehlszeiger. *BC* darf deshalb entweder durch den Assemblertext nicht verändert werden, oder sein Inhalt muß wieder rekonstruiert werden. Beim Code von TUCK wurde dies berücksichtigt.

```
CODE TUCK      ( ps: 16b1 16b2 ===>        )
               (           16b2 16b1 16b2 )
     HL POP,          ( lesen 16b2         )
     DE POP,          ( lesen 16b1         )
     HL PUSH,         ( schreiben 16b2     )
     DE PUSH,         ( schreiben 16b1     )
     HL PUSH,         ( schreiben 16b2     )
     NEXT # JP,       ( nächster Befehl )
     END-CODE
```

Bild 4.10 Assemblerimplementierung von TUCK

Im Gegensatz dazu darf das Register *DE*, das als Wortzeiger benutzt wird, zerstört werden, da es immer wieder neu gesetzt wird. Das Sprungziel NEXT sollte der Assembler als symbolische Bezeichnung bereitstellen. *wird fortgesetzt*

Tafel 4.3 Kurzbeschreibung der Forthworte

Name	Stapeleffekt	Beschreibung
Wortstruktur		
>BODY	('cf ===> 'pf)	Umrechnung der Code- in die Parameterfeldadresse
>LINK	('cf ===> 'lf)	Umrechnung der Code- in die Verkettungsfeldadresse
>NAME	('cf ===> 'nf)	Umrechnung der Code- in die Namensfeldadresse
BODY>	('pf ===> 'cf)	Umrechnung der Parameter- in die Codefeldadresse
L>NAME	('lf ===> 'nf)	Umrechnung der Verkettungs- in die Namensfeldadresse
LINK>	('lf ===> 'cf)	Umrechnung der Verkettungs- in die Codefeldadresse
N>LINK	('nf ===> 'lf)	Umrechnung der Namens- in die Verkettungsfeldadresse
NAME>	('nf ===> 'cf)	Umrechnung der Namens- in die Codefeldadresse
Suchen und indirekter Aufruf		
'	(===> 'cf) ib: name	Ermittelt die Codefeldadresse des Worts, dessen Name im Eingabepuffer folgt (ib – input buffer).
[']	(===>) ib: name	Analog zu ', nur zur Übersetzungszeit. Die Codefeldadresse wird als Literal gespeichert und zur Laufzeit auf dem Stapel übergeben.
EXECUTE	('cf ===>)	Das durch die Codefeldadresse gegebene Wort wird ausgeführt.
FIND	(addr ===> 'cf n) oder (addr ===> addr 0)	Die Kette ab addr wird im Wörterbuch gesucht. Falls ein Wort dieses Namens existiert, wird addr durch die Codefeldadresse ersetzt, n ist 1 für bevorzugte Worte, sonst – 1. Falls das Wort nicht gefunden wird, liegt über addr *falsch*.
Virtueller Prozessor		
>R	(16b ===>) rs: ===> 16b	Transport von 16b auf den Rückkehrstapel
R>	(===> 16b) rs: 16b ===>	Rücktransport von 16b vom Rückkehrstapel
R@	(===> 16b) rs: 16b ===> 16b	Kopie der Rückkehrstapelspitze
BRANCH	(===>)	Unbedingter Sprung, das Ziel wird im Fadencode übergeben
?BRANCH	(? ===>)	Sprung falls Null, sonst wie **BRANCH**
Mikroprogrammierung		
ASSEMBLER	(===>)	Vokabular zur Aufnahme des Assemblerwortschatzes
CODE	ib: name	Definionswort für ein Primärwort, dessen Name im Eingabepuffer folgt. Muß durch **END-CODE** abgeschlossen werden.
END-CODE	(===>)	Abschluß einer Primärdefinition

Einführung in Forth-83

Teil 5

Dr. Hartmut Pfüller (Leiter),
Dr. Wolfgang Drewelow, Dr. Bernhard Lampe
Ralf Neuthe, Egmont Woitzel
Wilhelm-Pieck-Universität Rostock,
Sektion Technische Elektronik

5. Die qualitative Erweiterung von Forth

Beim Vergleich mit anderen Programmiersprachen fällt auf, daß im Kern von Forth keine Definitionsworte für Datenstrukturen wie Felder oder Records enthalten sind. Die Bedeutung von Datenstrukturen ist aber für die Programmierung unbestritten, so daß die Frage steht, wie solche Probleme in Forth rationell zu behandeln sind.
Anstatt eine mehr oder weniger fest vorgefertigte Menge von starren Datentypen vorzugeben, stellt Forth dem Programmierer Werkzeuge zum Eigenbau von Datenstrukturen zur Verfügung. Er kann damit seine Datenstrukturen für das Problem maßschneidern und muß nicht das Problem solange umformen, bis es in eine angebotene Datenstruktur paßt.
Zu dieser für Forth typischen Vorgehensweise gehört auch die Möglichkeit, spezielle Werkzeuge zu schaffen, die eine rationelle und problemnahe Behandlung der selbsterzeugten Datenstrukturen sichern. Welchen Weg schlägt Forth ein, um dieses Ziel zu erreichen? Das Prinzip ist vergleichbar mit der Vorgehensweise bei der im Teil 3 beschriebenen Ausgabeformatierung. Auch die komplexen Definitionsworte wie **:** oder **VARIABLE** basieren auf elementaren Bestandteilen, die dem Programmierer sämtlich einzeln zugänglich sind und die zur Konstruktion neuer definierender Worte benutzt werden können.

5.1. Die Verwaltung des Wörterbuchspeichers

5.1.1. Funktionen

Eine Orientierung darüber, wo für den nächsten Wörterbucheintrag freier Speicherplatz ist, erhält man dadurch, daß das Wort **HERE** die entsprechende Adresse zum Stapel schafft. Wenn diese Adresse in einer (nicht standardisierten) Variablen **DP** (Directionary Pointer, deutsch: Wörterbuchzeiger) geführt wird, könnte HERE so implementiert sein:

```
: HERE ( ===> addr) DP @ ;
```

Um das Wörterbuch am Ende zu verlängern, kann das Wort **ALLOT** benutzt werden. ALLOT erwartet eine Zahl auf dem Datenstapel und reserviert dem Wert entsprechend viele Bytes für das Wörterbuch von der durch HERE bezeichneten Stelle ab. Der Inhalt der mit ALLOT reservierten Speicherplätze wird nicht gesetzt, er bleibt so, wie er sich zufällig bis zu diesem Zeitpunkt ergeben hatte. Systemintern könnte ALLOT etwa so definiert sein:

```
: ALLOT ( n ===> ) DP +! ;
```

Ebenfalls eine Vergrößerung des Wörterbuchs erreicht man mit den beiden Worten

,   C,

Das Wort **,** schreibt die auf dem Datenstapel befindliche 16-Bit-Zahl ans Wörterbuch hintan; das Wörterbuch wird damit um 2 Byte länger, wobei der Inhalt dieser 2 Byte durch die vom Stapel geholte Zahl gesetzt wird. Entsprechend vergrößert **C,** das Wörterbuch um genau ein Byte, wobei die 8 Bit, die dessen Inhalt bestimmen, auch wieder als Wert vom Stapel geholt werden. Die beiden Worte könnten im System beispielsweise so definiert sein:

```
: , ( 16b ===> ) HERE 2 ALLOT ! ;
: C, ( 8b ===> ) HERE 1 ALLOT C! ;
```

Beim Zugang zur physischen Ebene müssen von Fall zu Fall spezifische Schaltkreiseigenschaften berücksichtigt werden. Angenommen, der Programmierer wünscht, ins Wörterbuch eine Tabelle von Zeigern zu Interruptserviceroutinen einzutragen, und der Prozessor verlangt, daß die Tabelle bei einer durch 8 teilbaren Adresse beginnt. Die Variable DP kann dann durch eine Wortfolge wie

```
HERE NEGATE 8 MOD ALLOT
```

auf die nächste durch 8 teilbare freie Adresse des Wörterbuchs gesetzt werden. Mittels solcher und ähnlicher Techniken kann man beispielsweise die Interruptbehandlung vollständig von Forth aus organisieren.

5.1.2. Vektoren und Tabellen

Die genannten Mittel gestatten bereits die Vereinbarung von Datenbereichen im Wörterbuch. Beispielsweise kann durch die Sequenz

```
VARIABLE ZAEHLERFELD 8 ALLOT
```

ein Feld für 5 Werte von je 16 Bit im Wörterbuch reserviert werden; ein Wert direkt durch **VARIABLE** und die restlichen vier Werte durch **8 ALLOT**. **VARIABLE** legen bei der Definition von **ZAEHLERFELD** dessen Laufzeitverhalten fest: Beim Aufruf von ZAEHLERFELD wird die Basisadresse (die Adresse der 0. Komponente) auf den Datenstapel gelegt. Die übrigen Komponenten sind durch den Offset von 2, 4, 6, 8 adressierbar, für Index 3 etwa so:
ZAEHLERFELD 6 +
Die Initialisierung aller fünf Zähler mit Null kann nach

```
: RUECKSETZEN ( 'feld ===> )
  5 0 DO 0 OVER ! 2+
    LOOP DROP ;
```

durch die Wortfolge
ZAEHLERFELD RUECKSETZEN
gemeinsam ausgeführt werden. Durch die Zusammenfassung von Zählern in einem Feld können solche Operationen vereinfacht werden, die alle Zähler des Feldes betreffen. Dazu gehören neben der Initialisierung beispielsweise Vergleiche der Zählerstände oder deren Auswertung und Darstellung.
Im obigen Beispiel ist es eventuell nachteilig, daß die Initialisierung nur für Felder der festen Dimension 5 arbeitet. Will man Felder variabler Länge in ähnlicher Weise behandeln, so muß die zugeordnete Feldlänge ermittelt werden können. Das läßt sich erreichen, wenn die Felddimension mit abgespeichert wird (beispielsweise vor dem Feld selbst). Entsprechend der oben stehenden Variante könnte man

```
VARIABLE ZAEHLERFELD 10 ALLOT
5 ZAEHLERFELD !
```

schreiben. Noch einfacher ist jedoch die Konstruktion, wenn man als Definitionswort für den Wörterbucheintrag das (primitivere) Definitionswort **CREATE** verwendet. Es wirkt wie VARIABLE, reserviert selbst aber nach dem Namen noch keinen zusätzlichen Speicherplatz für Daten. Die Folge

```
CREATE ZAEHLERFELD 5 , 10 ALLOT
```

spiegelt die Struktur des Feldes gut wider; CREATE legt das Laufzeitverhalten von ZAEHLERFELD fest. Das mit

```
: INITIALISIEREN ( 'feld ===> )
  DUP @ 0 DO 2+ 0 OVER !
    LOOP DROP ;
```

definierte Wort initialisiert so ein Feld variabler Länge.
Fließpunktarithmetik und transzendente Funktionen sind nicht Bestandteil des Forthkerns, weil sich die Anforderungen der Anwender an Genauigkeit und Verarbeitungsgeschwindigkeit stark unterscheiden. Außerdem lassen sich viele Probleme mit ganzen Zahlen lösen, wenn man auf die (gefährliche) Bequemlichkeit des Rechnens ohne vorherige Abschätzung verzichtet. Als Beispiel soll die Funktionsberechnung unter Verwendung von Funktionstafeln erläutert werden, wobei man bei geeigneter Skalierung im Bereich der ganzen Zahlen bleibt.
Eine mit

```
CREATE SINUSTAFEL
      0 ,  175 ,  349 ,  523 ,  698 ,
    872 , 1045 , 1219 , 1392 , 1564 ,
   1737 , 1908 , 2079 , . . .
   .
   .
   .     9976 , 9986 , 9994 , 9999 ,
  10000 ,
```

erzeugte Tafel enthält die mit 10000 skalierten Sinuswerte zu den Winkeln von 0, 1, ..., 90 Grad. Hier wird die Zweckmäßigkeit der Vergabe des Namens **,** für die direkte Eintragung ins Wörterbuch erkennbar. Ein Leser des Programmteils SINUSTAFEL muß nicht unbedingt mit dem an das Komma gebundenen Kompilationsvorgang vertraut sein; es genügt, die im gewöhnlichen Sprachgebrauch übliche Bedeutung des Kommas als Trennsymbol zu kennen. Durch die ähnlich geschickte Wahl der Namen kann der Programmierer die Lesbarkeit seiner Programme günstig beeinflussen. Das Aufsuchen des passenden Sinuswertes geschieht nach

```
: SINUS ( n ===> sin[n] )
  2 * SINUSTAFEL + @ ;
```

in der Form
45 SINUS . (cr) 7071 ok

Unter Ausnutzung der Periodizität können beliebige Winkel auf den angegebenen Argumentbereich zurückgerechnet werden. Durch einen Suchprozeß innerhalb der Tabelle ist auch die Arkussinusfunktion implementierbar. Das oben definierte Wort **SINUS** stellt die primitivste Form des Zugriffs auf die Tabelle dar. Durch lineare Interpolation ließen sich auch Zwischenwerte berechnen.

5.2. Definition von Wortklassen

Wenn mehrere Objekte die gleiche oder eine ähnliche Struktur haben, kann man, wie am Beispiel der Initialisierung deutlich wurde, Operationen so erzeugen, daß sie über allen Objekten einer solchen Familie ausführbar sind. Wünschenswert wäre nun, auch die Kompilationsvorschrift für zur gleichen Familie gehörende Objekte nur einmal erklären zu müssen. Das spart Programmspeicherplatz und erhöht die Übersichtlichkeit. Mit der einmal erzeugten Form können dann viele gleichartige Objekte „gegossen" werden.
Die Konstruktion einer „Gußform" wird ermöglicht, weil auch Definitionsworte (z. B. CREATE) selbst wieder in Definitionen verwendet werden dürfen. Auf diese Art kann der Anwender dann eigene Definitionsworte definieren, die bei ihrer Ausführung selbst wieder neue Worte eines (vom Anwender) bestimmten Typs erzeugen. So können mit einem vom Programmierer neu geschaffenen Definitionswort spezielle Eigenschaften für eine ganze Klasse von neuen Worten fixiert werden. Die zweckmäßige Festlegung von problemgerechten Wortklassen kann in hohem Maße die Qualität eines Programms bestimmen. Sie verlangt eine gründliche Analyse der Problemstellung, insbesondere unter dem Gesichtspunkt der Zerlegung in ähnlich geartete Teilprobleme. Nach Erklärung des Datentyps

```
: FELD ( n ===> )
  CREATE DUP , 2 * ALLOT ;
```

beispielsweise würde durch

5 FELD ZAEHLERFELD (===> addr)

ein Zählerfeld der Dimension 5 ins Wörterbuch eingetragen werden. FELD ist nun das Definitionswort für einen vom Programmierer neu geschaffenen Datentyp. Das Laufzeitverhalten von ZAEHLERFELD wird wiederum durch CREATE festgelegt. Nach Aufruf von ZAEHLERFELD würde die Parameterfeldadresse des ZAEHLERFELDes auf den Datenstapel gelegt werden; dort war bei der Definition durch FELD die Dimension eingetragen worden.

5.2.1. *Fadencodeniveau*

Mitunter wird es sinnvoll sein, wenn nach dem Aufruf eines mit FELD erzeugten Objektes nicht die Adresse der Dimension, sondern die Adresse des ersten Feldelements zum Stapel geliefert wird. Das Laufzeitverhalten, das dem neudefinierten Wort durch CREATE mitgegeben wurde, müßte zu diesem Zweck anders programmiert werden können. Speziell für diesen Zweck ist das Wort **DOES>** vorgesehen. Die in der Konstruktion

: name . . . CREATE . . . DOES> . . . ;

nach DOES> stehenden Worte werden bei Aufruf jedes mit *name* definierten Objektes ausgeführt. Dabei ist es wichtig zu wissen, daß unmittelbar vorher die Parameterfeldadresse des Objekts auf dem Stapel bereitgelegt wird. Die auf DOES> folgenden Worte können dann diese Information über den Beginn des Parameterfeldes verwenden, um damit ein gewünschtes Laufzeitverhalten des Objekts zu erzeugen.
Eine neue Klasse von Datenfeldern, deren Abkömmlinge zwar ihre Dimensionen enthalten, aber beim Aufruf die Adresse ihrer nullten Komponente auf den Stapel legen, wird mit

```
: VEKTOR ( n ===> )
  CREATE DUP , 2* ALLOT
  DOES> ( ===> 'x0 ) 2+ ;
```

beschrieben. Beim Stapelkommentar nach DOES> muß man immer berücksichtigen, daß unmittelbar vor dem Abarbeiten der DOES>-Passage noch die Parameterfeldadresse des definierten Objekts obenauf gelegt wird. Sie sollten etwas Zeit investieren, um sich klarzumachen, welche Konsequenzen es hat, daß die Passage hinter DOES> *nicht* bei der Ausführung von VEKTOR ausgeführt wird. Vielmehr sorgt DOES> dafür, daß bei jeder Definition eines Objekts mittels **VEKTOR** das Codefeld dieses Objekts mit einem Verweis zur DOES>-Passage überschrieben wird. Die Worte nach DOES> werden damit genau bei jedem Aufruf eines Objekts abgearbeitet. Dadurch kommt auch die Familieneigenschaft aller mit demselben Definitionswort erzeugten Objekte zustande.
Der Zugriff auf Komponenten von mit VEKTOR definierten Vektoren wie

5 VEKTOR ZAEHLERFELD (===> addr)
ließe sich durch

```
: KOMPONENTE ( 'x n ===> 'xn)
  2 * + ;
```

bewerkstelligen. So wird durch
ZAEHLERFELD 3 KOMPONENTE @
der Inhalt des Elements mit dem Index 3 auf den Datenstapel gelegt.
Die Konstruktion **CREATE . . . DOES>** macht es möglich, die Kompilationsphase für Objekte einer Wortklasse vollständig von der Phase der Ausführung getrennt zu betrachten und zu programmieren. Der Programmierer kann die aus seiner Sicht insgesamt erforderlichen Aktionen zerlegen und der jeweils entsprechenden Phase zuordnen (für den Kompilationsvorgang hinter CREATE und für den Ausführungsvorgang hinter DOES>).
Zur Erläuterung des Gebrauchs von Wortklassen soll nun ein Beispiel aus Teil 2 des Kurses etwas modifiziert behandelt werden: In einer Variablen MATERIAL stehe (wie auch damals) die Adresse des Zählers für Teile des gerade aktuellen Materials. Die Reservierung von Speicherplatz für einen materialabhängigen Zähler und die Bereitstellung seiner Adressen bei Nennung des Materials werden jetzt in dem Definitionswort

```
: MATERIAL:
  VARIABLE ( installiert Zähler)
  DOES> MATERIAL ! ( lädt Zähler) ;
```

zusammengefaßt. Nach den Definitionen
MATERIAL: HOLZ
MATERIAL: PLAST
und den „Verschönerungen"

```
: DAZU ( n addr ===> ) +! ;
: ? ( addr ===> ) @ . ;
```

sind die in Teil 2 benutzten Aktionen wie
7 HOLZ TEILE DAZU
PLAST TEILE ?
wieder verwendbar.
Will man verschiedenartige Teile nach Material sortiert zählen, so kann man für jedes Material mehrere „Körbe", das heißt ein Zähler*feld* bereitstellen. Der zum jeweiligen Teil gehörende Korb ist durch einen Offset zum Anfang des Zählerfeldes des entsprechenden Materials erreichbar (Bild 5.1).
Das Definitionswort TEIL: speichert während der Definition eines Teils den vom Programmierer zugeordneten Offset n verdoppelt ab

```
VARIABLE  MATERIAL

:  TEIL:          ( n ===> )
   2  *  CONSTANT
         DOES>    ( ===> korbadresse)
            @   MATERIAL  @  +  ;

2  CONSTANT   #TEILE
0  TEIL:      KEILE
1  TEIL:      STIFTE

:  MATERIAL:  ( ===> )
   CREATE  #TEILE   0  DO  0  ,  LOOP
   DOES>      ( ===> )   MATERIAL  !  ;

MATERIAL:  HOLZ
MATERIAL:  PLAST
```

Bild 5.1 Beispiel zu CREATE . . DOES>

und bildet bei Nennung des jeweiligen Teils die Adresse des zugehörigen materialrichtigen Zählers. Damit ist die Schreibweise

HOLZ 12 KEILE DAZU 60 STIFTE DAZU PLAST STIFTE ?

möglich. Wenn für #TEILE vorsorglich ein etwas größerer Wert vereinbart wird, können mittels TEIL: auch noch nachträglich weitere Arten von Teilen definiert werden. Die Konstruktion CREATE . . . DOES> ist einfach, leistungsfähig und ungewöhnlich. Wir empfehlen Ihnen, eigene Experimente mit betont unkomplizierten Schritten zu beginnen. Eventuell sind in der ersten Zeit erläuternde Hilfsausschriften nützlich, wie im folgenden Beispiel für ein Definitionswort, das Objekte erzeugen kann, die nichts anderes tun, als ihre eigenen Aktivierungen mitzuzählen:

```
: ZAEHLER ." vor Definition "
  CREATE  ." nach Definition " 0 ,
  DOES>   ." bei Aktivierung"
    1 OVER +! @ . ;
```

Nach der Erzeugung eines Objekts mit
ZAEHLER KLICK
zeigt KLICK bei jeder Aktivierung an, der wievielte Aufruf das war.

5.2.2. *Mikrocodeniveau*

Alternativ zum Fadencode kann der Ausführungsteil auch in Maschinencode notiert werden. Für diesen Fall ist das Wort **;CODE** anstelle von **DOES>** zu benutzen. Die auf ;CODE folgenden Worte müssen dann Be-

standteil des Forth-Assemblers sein, der vorher geladen worden sein muß. Falls kein Assembler verfügbar ist, kann auch der von Hand übersetzte Maschinencode direkt mit Hilfe von **,** oder **C,** eingetragen werden. Man wird ;CODE benutzen, wenn man auf die größere Rechengeschwindigkeit von Assemblerprogrammen angewiesen ist. Zum Beispiel könnte man die Zugriffszeit auf Werte in Tabellen verkürzen, wenn man sie mit dem Definitionswort nach Bild 5.2 erzeugt. Der Programmierer braucht auch hier einen Zugang zum Parameterfeld der letztlich zu definierenden Objekte. Während DOES> hierfür die Parameterfeldadresse (PFA) auf dem Stapel hinterläßt, regelt sich nach ;CODE der Zugang zur PFA des definierten Objekts über das WA-Register der virtuellen Forthmaschine. Welches Prozessorregister in einer konkreten Implementation dafür verwendet wird, muß der zugehörigen Dokumentation entnommen oder anderweitig in Erfahrung gebracht werden. Selbstverständlich hat man sich mit der hier vorgeführten Version für das Definitionswort TAFEL auf den Prozessor Z 80 und meist auch noch auf einen bestimmten Assembler festgelegt.

```
:  TAFEL        ( xn-1 . . . x1 x0 n ===> )
   CREATE
        0  DO  ,  LOOP
   ;CODE       ( i ===> xi )
   DE       INC,                      ( WA-Register auf PFA justieren)
   HL       POP,                      ( Index i vom Stapel holen)
   HL HL    ADD,  DE HL  ADD,         ( Elementadresse berechnen)
   (HL)  E  LD,   HL     INC,         ( Element nach DE laden)
   (HL)  D  LD,
   DE      PUSH,                      ( Element xi zum Stapel)
   NEXT  #  JP,                       ( zurueck zum Adressinterpreter)
   END-CODE
```

Bild 5.2 ***Beispiel zu CREATE...;CODE***

Nach der Vorbereitung

10000 9999 . . . 175 0
91 TAFEL SINUS

kann die Sinusfunktion für einen bestimmten Winkel wieder mit

n SINUS

ermittelt werden. Solche Tafeln sind ein sehr bequemes und rechenzeiteffektives Verfahren, um Funktionen implementieren zu können.

5.3. Compilererweiterungen

5.3.1. Vorrangworte

Im Teil 2 des Kurses (s.MP 5/89) wurden Strukturierungsworte vorgestellt. Sie dienen im Kompilationsmodus zum Aufbau eines möglichst laufzeitrationellen Fadencodes und werden deshalb innerhalb einer Definition *ausgeführt*, während „normale" Worte im Gegensatz dazu in Fadencode *kompiliert* werden. Soll ein bestimmtes vom Programmierer definiertes Wort diese besondere Eigenschaft erhalten, so daß es trotz des Kompilationsmodus ausgeführt wird, so ruft man nach dessen Definition das Wort **IMMEDIATE** (deutsch sinngemäß: unverzüglich) auf. Das zuletzt definierte Wort wird dadurch zu einem *Immediate*-Wort oder *Vorrangwort.*

Die Ausführung von Vorrangworten wird gewissermaßen in die Kompilationszeit vorgezogen.

Das bisher betont einfache Prinzip der Kompilation von neuen Worten durch Zusammenstellung von Worten im Quelltext und durch – dafür jeweils eins zu eins – deren Codefeldadressen im Kompilat kann mittels der Vorrangworte durchbrochen werden, weil sie in der Regel nicht ins Wörterbuch gelangen und stattdessen sogar andere Worte ins Wörterbuch kompilieren können. Das würde der Programmierer zunächst gar nicht bemerken, wenn er von anderen Programmiersprachen her gewöhnt ist, daß ihm der Zugang zur Gestaltung des Kompilats in der Regel verschlossen bleibt.

Der versierte Forthprogrammierer wird von Fall zu Fall wünschen, Vorrangworte nicht zur Kompilationszeit arbeiten zu lassen, sondern sie entgegen ihrem Normalverhalten trotzdem zu kompilieren. Dafür gibt es die Möglichkeit, die Vorrangeigenschaft fallweise explizit zu unterdrücken.

Mit dem Wort **[COMPILE]** kann man die Kompilation eines Wortes erzwingen; die eventuelle Vorrangeigenschaft wird unterdrückt. Die erzwungene Kompilation betrifft nur das unmittelbar auf [COMPILE] folgende Wort. Damit [COMPILE] diese Wirkung erzielen kann, ist es selbst als Vorrangwort erklärt.

Als Gegenstück zur vorgezogenen Ausführung von Worten ist auch eine aufgeschobene Kompilation möglich. Das erreicht man durch eine Wortfolge der Form

: yyy . . . COMPILE xxx . . . ;

Der Effekt ist der, daß yyy bei seiner Ausführung eine Kompilation von xxx durchführt (also die Eintragung der Codefeldadresse von xxx ans Wörterbuchende). Während der Definition von yyy werden die CFAs von **COMPILE** und anschließend von xxx ganz normal ins Wörterbuch eingetragen. Beim Aufruf von yyy sorgt dann COMPILE dafür, daß die CFA von xxx noch einmal an das aktuelle Ende des Wörterbuchs kompiliert wird. Typisch wird COMPILE in Vorrangworten verwendet, wie wir in einem Beispiel im nächsten Abschnitt zeigen werden.

5.3.2. Programmverzweigungen

Im Abschnitt 2.3 wurden Worte gezeigt, mit denen Programme nach den Regeln der strukturierten Programmierung gegliedert werden können. Bei der Erzeugung *eigener* Verzweigungsstrukturen ist es zuweilen effektiver, sie aus elementaren Bestandteilen neu aufzubauen, anstatt die aus dem Anwenderwortschatz geläufigen vorgefertigten Strukturierungsworte zu verwenden.

Zu den elementaren Bestandteilen der Verzweigungsstrukturen gehören die Worte

>MARK >RESOLVE <MARK <RESOLVE

Mit ihnen werden zur Kompilationszeit Sprungadressen in Verzweigungsstrukturen verwaltet. Es gehören jeweils eine Markierung (MARK) und eine Auflösung (RESOLVE) zusammen. Mit MARK wird eine Marke (Adresse) auf den Datenstapel gelegt, zu der später von derjenigen Position des Wörterbuchzeigers aus gesprungen werden muß, die bei Nennung von RESOLVE gerade aktuell war (oder wird). Die Richtung der Sprünge ist an den Namen MARK und RESOLVE durch < für rückwärts und > für vorwärts abzulesen.

Besonders einfach ist die Eintragung von Rückwärtssprüngen durch

: <MARK (===> addr) HERE ;
: <RESOLVE (addr ===>) , ;

nachvollziehbar. Bei Vorwärtsreferenzen wird die Zieladresse erst zu einem späteren Zeitpunkt ermittelt, so daß der Speicherplatz erst reserviert und später die Adresse dort eingetragen wird. Auch hier läßt sich der möglichen Implementierung

: >MARK (===> addr)
HERE 2 ALLOT ;
: >RESOLVE (addr ===>)
HERE SWAP ! ;

die Arbeitsweise dieser Worte entnehmen. Da die Worte nur die Eintragung der Sprungadressen leisten, müssen sie noch mit Sprungbefehlen und Sprungbedingungen verknüpft werden (vergl. BRANCH, ?BRANCH in Teil 4 des Kurses).

Die Sprungadreßberechnungen sind so angelegt, daß sie sich verschachteln lassen. Damit dem Programmierer dabei keine Flüchtigkeitsfehler unterlaufen, prüfen die meisten Forth-Systeme während der Kompilation die Vollständigkeit der Verzweigungsstrukturen. Oft wird dazu bei Eröffnung der Verzweigung über die Adresse eine Kennzahl gelegt, deren Anwesenheit im Schließbefehl vor Ausführung der Adreßverknüpfung überprüft wird. Auf solche einfachen Sicherungsmaßnahmen sollte man bei eigenen Kompilationsworten nicht verzichten. Man kann eigene Kompilationsworte gerade mit dem Ziel schaffen, während der Kompilation weitgehend die Richtigkeit des Programms zu überprüfen.

Als Beispiel für neue Kompilationsworte soll eine inzwischen weit verbreitete Fallkonstruktion betrachtet werden. Sie verallgemeinert die Struktur **IF . . . ELSE . . . THEN** auf eine beliebige Anzahl von Fällen. Welcher Zweig des Programms abgearbeitet wird, soll sich aus dem Wert einer zur Abarbeitungszeit auf dem Datenstapel liegenden Zahl (Fallnummer) ergeben. Die Struktur wird in der Weise

```
: . . . CASE n1 OF . . . ENDOF
             n2 OF . . . ENDOF
              .   .   .   .
             nk OF . . . ENDOF
  . . . ENDCASE . . . ;
```

benutzt. Die vorkommenden Worte können wie in Bild 5.3 definiert werden. Alle vier definierten Strukturierungsworte werden jeweils durch IMMEDIATE zu Vorrangworten erklärt; mit COMPILE werden die erforderlichen Wörterbucheinträge erzwungen.

Die Kompilation eines die **CASE**-Konstruk-

```
:  CASE  ( ===> addr 5 ; Strukturanfang )
   CSP  @  SP@  CSP  !  5  ;            IMMEDIATE

:  OF     ( 5 ===> addr 6 ; Zweiganfang )
   5  ?PAIRS  ( Struktursicherung )
   COMPILE  OVER    COMPILE  =
   COMPILE  ?BRANCH
   >MARK   COMPILE DROP   6  ;          IMMEDIATE

:  ENDOF  ( addr1 6 ===> addr2 5 ; Zweigende)
   6  ?PAIRS   ( Struktursicherung )
   COMPILE  BRANCH
   >MARK  SWAP  >RESOLVE  5 ;           IMMEDIATE

:  ENDCASE  ( addr addr1 ... addrk ===>)
   5  ?PAIRS   ( Struktursicherung )
   COMPILE  DROP
   ( Spruenge ans Strukturende nachtragen:)
     BEGIN  SP@  CSP  @  -  WHILE  >RESOLVE
     REPEAT
   CSP ' ; ( CSP wiederherstellen)  IMMEDIATE
```

Bild 5.3 Codierung einer CASE-Verzweigung

tion enthaltenden Wortes soll nun kurz verfolgt werden:
Das Wort CASE eröffnet die Struktur, indem es den Wert der Variablen **CSP** auf den Stapel rettet, den aktuellen Stand des Stackpointers für einen späteren Vergleich in der Variablen CSP speichert und die willkürlich gewählte Kennzahl 5 zur späteren Kontrolle der Vollständigkeit der Struktur auf den Stapel legt. Danach wird die Fallnummer ins Wörterbuch kompiliert. Das darauf folgende **OF** kontrolliert die richtige Reihenfolge innerhalb der CASE-Konstruktion, indem die durch CASE auf dem Stapel hinterlassene 5 abgefragt wird. Falls **?PAIRS** nicht bekannt ist, kann man vorläufig

```
: ?PAIRS ( n1 n2 ===> )
  - IF ." falsche Struktur " THEN ;
```

definieren. Normalerweise sollte man allerdings abbrechen, wenn Strukturfehler erkannt worden sind. Im Teil 6 des Kurses wird auf Möglichkeiten zur Fehlerbehandlung eingegangen.
Anschließend werden mit der Wortfolge **OVER = ?BRANCH**, der Platz für die Sprungadresse und **DROP** kompiliert. Zum Abschluß von OF wird die Kennzahl 6 auf den Datenstapel gelegt. Nun folgt die Kompilation des Zweigprogramms. Danach muß der Datenstapel denselben Zustand haben, damit die Prüfung der Kennzahl 6 durch **ENDOF** positiv ausfällt. Von ENDOF wird außerdem der mit OF kompilierte bedingte Vorwärtssprung auf den nach ENDOF vorzunehmenden Wörterbucheintrag, das heißt die nächste Fallabfrage, gerichtet. Der Speicherplatz für die Adresse des im Falle der Übereinstimmung notwendigen unbedingten Sprungs ans Ende der Gesamtkonstruktion wird durch das in ENDOF stehende >**MARK** reserviert.
Das Wort **ENDCASE** löst die von den ENDOFs auf dem Datenstapel hinterlassenen Markierungsadressen (für die unbedingten Sprünge ans Ende der Konstruktion) auf. Durch Vergleich des Stapelzeigers mit dem in CSP abgelegten alten Wert wird geprüft, ob die richtige Anzahl von Adressen zugeordnet wurde.
Wir empfehlen Ihnen, die Kompilation und die Ausführung des folgenden, als Demonstration gedachten Wortes nachzuvollziehen:

```
: .X (n ===> )
  CASE 0 OF ." Zweig 0 " ENDOF
       1 OF ." Zweig 1 " ENDOF
       2 OF ." Zweig 2 " ENDOF
          ." Zweig x" ENDCASE ;
```

Tafel 5 Kurzbeschreibung der Forthworte

Name	Stapeleffekt	Beschreibung
Wörterbuchverwaltung		
,	(16b ===>)	16b als neuen Eintrag ans Wörterbuch anhängen
C,	(8b ===>)	8b als neuen Eintrag ans Wörterbuch anhängen
ALLOT	(n ===>)	reserviert die nächsten n Byte im Wörterbuch
HERE	(===> addr)	addr ist die nächste freie Adresse am Wörterbuchende
IMMEDIATE	(===>)	erklärt das zuletzt definierte Wort zum Vorrangwort
Definitionsworte		
CREATE	(===>)	erzeugt den Kopf für eine Variable
DOES>	(===> pfa)	Beginn des für alle Elemente der Wortklasse gleichen Laufzeitcodes (Hochcode)
:CODE	(===>)	ähnlich wie DOES>, aber es folgt Assembler- oder Maschinencode
Kompilationsworte		
<MARK	(===> addr)	liefert die aktuelle Wörterbuchendeposition addr zum Stapel
<RESOLVE	(addr ===>)	benutzt die addr von <MARK zur Berechnung und Eintragung der Adresse eines Rückwärtssprunges
>MARK	(===> addr)	liefert die aktuelle Wörterbuchendeposition addr und reserviert Platz für eine Vorwärtssprungadresse
>RESOLVE	(addr ===>)	bildet aus der von >MARK gelieferten addr und der aktuellen Wörterbuchendeposition einen Vorwärtssprung und trägt diesen in den von >MARK reservierten Platz ein
COMPILE	(===>)	kompiliert in der Ausführungsphase das nachfolgende Wort
LITERAL	(Kompilation: n ===>) (Ausführung: ===> n)	kompiliert die auf dem Parameterstack liegende Zahl als Literal und liefert diese bei der Ausführung zum Stapel
STATE	(===> addr)	Variable, die während des Ausführungsmodus den Wert Null hat
[	(===>)	schaltet den Ausführungsmodus ein
]	(===>)	schaltet den Kompilationsmodus ein
[COMPILE]	(===>)	kompiliert das im Quelltext nächstfolgende Wort; auch dann, wenn es ein Vorrangwort ist

5.3.3. Zustandssteuerung

Grundsätzlich befindet sich das Forthsystem entweder im Modus *Kompilieren* oder im Modus *Ausführen*. Welcher Zustand gerade aktiv ist, kann der Systemvariablen **STATE** entnommen werden. Im Modus Ausführung hat die Variable STATE den Wert Null, ansonsten ist sie davon verschieden (implementationsabhängig). STATE ist für die Abfrage des Systemzustandes vorgesehen und darf vom Anwender nicht selbst verändert werden. Manchmal kommen in Programmen Rechenoperationen mit solchen Operanden vor, deren Werte schon zur Übersetzungszeit bekannt sind. In solchen Fällen wäre es günstig, die entsprechenden Operationen tatsächlich schon zur Übersetzungszeit zu erledigen, so daß im übersetzten Programm schon die fertigen Zwischenergebnisse verwendet werden. Die aufgewendete Rechenzeit spart man bei der wiederholten Abarbeitung des definierten Wortes mehrfach ein. Die Berechnungen erfordern den Zustand *Ausführung*, der während der Kompilation vorübergehend eingeschaltet werden müßte. Nach Berechnung des Wertes wird dann weiter kompiliert. Zur Umschaltung des Zustandes werden die beiden Worte

[ ]

benutzt. Die Notation läßt sich leicht merken, weil die in Klammern stehenden Ausdrücke als „Nebenrechnung" aufgefaßt werden können, die nicht kompiliert wird. Als Ergebnis der „Nebenrechnung" wird in der Regel ein Zahlenwert auf dem Datenstapel geliefert. Seine Kompilation kann mit dem Wort **LITERAL** erfolgen, das einen zur Compilationszeit auf dem Datenstapel liegenden Wert ins Wörterbuch einträgt. Beim Aufruf der Passage wird der Wert auf den Stapel zurückgebracht.

Beispielsweise kann für ein Objekt der Klasse **VEKTOR** mit dem Namen **VX** nach Abschnitt 5.2.1. der Zugriff auf die dritte Komponente anstatt durch

```
: zzz... VX 3 KOMPONENTE...;
```

mit

```
: zzz... [ VX 3 KOMPONENTE ]
  LITERAL...;
```

vorgenommen werden. Der komplizierter aussehende zweite Quelltext stellt die günstigere Lösung dar. Statt der 4∗16 Bit Speicherplatz im Wörterbuch werden nur 2∗16 Bit benötigt. Außerdem wird während der Laufzeit des Wortes zzz die Passage schneller abgearbeitet, weil die in Klammern stehenden Operationen bereits während der Übersetzung in den Fadencode erledigt worden sind.
Die gewöhnlichen Zahlen werden übrigens während der Kompilation in derselben Weise behandelt. Zur Ablage einer Zahl im Kompilat werden deshalb in der Regel zweimal 16 Bit benötigt, einmal für LITERAL und einmal für die Zahl selbst. Das ist ein höherer Aufwand als für Konstanten, wo nur die Adresse (16 Bit) zur CFA der Konstanten ins Wörterbuch geschrieben wird. Damit gibt es einen weiteren Grund für die Verwendung von Konstanten.

wird fortgesetzt

Einführung in Forth-83

Teil 6

Dr. Hartmut Pfüller (Leiter), Dr. Wolfgang Drewelow, Dr. Bernhard Lampe, Ralf Neuthe, Egmont Woitzel, Rostock

6. Das Forth-Dialogsystem

6.1. Die Struktur des Dialogsystems

Zum Dialogsystem zählt man alle diejenigen Worte, die – grob gesagt – zum Komplex der Ablaufsteuerung für die Kommunikation gehören. Das sind der Kaltstart, der Textinterpreter und das Fehlersystem.

6.1.1. Initialisierung

Beim Start eines Forth-Systems muß zuerst dessen Initialisierung vorgenommen werden. Diese Aufgabe übernimmt in der Regel das Wort **COLD**. Dessen Anwesenheit ist im Standard Forth-83 zwar nicht vorgeschrieben, es ist aber in fast jedem System vorhanden. COLD ist der Hochwort-Eintrittspunkt für den Systemstart. Im wesentlichen werden damit die Speicheraufteilung festgelegt, das System nach den Festlegungen aus dem Boot-Bereich voreingestellt, die Stapel initialisiert, eine Systemausschrift zum Ausgabegerät gesendet und schließlich die Steuerung an den Textinterpreter übergeben.

6.1.2. Dialogführung

Der Dialog mit dem Bediener erfolgt unter Steuerung des Textinterpreters **QUIT**. Der Standard verlangt, daß QUIT den Rückkehrstapel säubert, den Ausführmodus einschaltet, die Eingabebereitschaft über das aktuelle Eingabegerät herstellt und die Wortinterpretation durchführt.

Innerhalb dieser Richtlinien kann der Rahmenablauf weitgehend variiert werden. Eine mögliche Variante ist im Bild 6.1 dargestellt. Hier wird QUIT mit der **DOER-MAKE**-Konstruktion /1/ vektoriert ausgeführt. Damit ist der gesamte Textinterpreter auch für Fachsprachen, die für Forth untypisch sind, komplett austauschbar.

6.1.3. Der Eingabestrom

Der Eingabestrom ist die Zeichenfolge, die vom Textinterpreter des Systems verarbeitet wird. Er kann sowohl vom aktuellen Eingabegerät (über den Texteingabepuffer) als auch vom Massenspeicher (über einen Blockpuffer) geholt werden. **BLK >IN TIB #TIB** spezifizieren den Eingabestrom. Worte, die auf BLK >IN TIB #TIB zugreifen oder deren Inhalte verändern, sind verantwortlich für das Aufrechterhalten (Sichern) und das Rückspeichern der Informationen zur Steuerung des Eingabestromes /2/.

Woher der Eingabestrom zu entnehmen ist, wird durch die Variable **BLK** spezifiziert. Sie enthält die Nummer des entsprechenden Massenspeicherblockes oder eine Null, wenn der Eingabestrom aus dem Texteingabepuffer kommen soll. Die Anfangsadresse dieses Texteingabepuffers kann mit **TIB** (Terminal-Input-Buffer) geholt werden. In der Variablen **#TIB** wird die aktuelle Anzahl der dort vorhandenen gültigen Bytes gehalten.

Die sequentielle Entnahme von Zeichenketten wird durch die Variable >**IN** unterstützt. Sie enthält den aktuellen Zeichenoffset zu der Stelle, an der die nächste Entnahme (aus dem Block- oder dem Texteingabepuffer) erfolgen kann. Bild 6.2 zeigt das Zusammenspiel der Worte.

WORD ist das zentrale Wort für die zerstörungsfreie Entnahme von Teilzeichenketten (typischerweise Namen oder Zahlen) aus dem aktuellen Eingabestrom. Dafür verwendet WORD ein Begrenzerzeichen, das ihm als Parameter auf dem Stapel übergeben wird. Vor der Kette befindliche Begrenzerzeichen werden übergangen, das entspricht einem Herantasten an den interessierenden Teil. Danach werden so lange Zeichen entnommen, wie das Begrenzerzeichen nicht angetroffen wird. WORD stellt dann die Variable >IN auf die Stelle im Eingabestrom ein, die unmittelbar hinter dem abschließenden Begrenzerzeichen liegt. So ist mit einer erneuten Ausführung von WORD die nächste Kette erreichbar. Ist der Eingabestrom erschöpft, liefert WORD eine leere Kette (Zählbyte = 0). WORD übergibt als Ergebnisparameter eine Adresse, die anzeigt, wo die separierte Kette im Standardformat abholbereit liegt.

6.1.4. Fehlerbehandlung

Das Fehlersystem von Forth achtet vor allem auf fatale Fehler wie Stapelunterlauf, verkehrte Verschachtelung von Steuerstrukturen, Wörterbuchüberlauf usw. Es schützt nicht vor Systemabstürzen infolge von liegengelassenen Parametern auf dem Rückkehrstapel oder nicht autorisiertem Überschreiben von Programmteilen.

Tests auf bestimmte Fehlerbedingungen können zu unterschiedlichen Zeitpunkten und in verschiedenen Worten stattfinden. Compilerworte wie **IF**, **ELSE** und **THEN** achten beispielsweise in vielen Systemen eigenständig auf ihre richtige Kombination.

Dem Anwender stellt Forth das Wort **ABORT"** zur Verfügung. Es wird in der Form

Flag ABORT" Fehlertext"

verwendet. Ist das Flag falsch, wird mit der Bearbeitung hinter ABORT"..." fortgesetzt. Ansonsten erfolgt die Ausgabe des Fehlertextes und der Eintritt in die Fehlerroutine ABORT.

6.2. Beispiel: Universeller Server und lernender Klient

Mit einem etwas größeren Programmbeispiel soll der Kurs Forth-83 nun beendet werden. Ausgewählt wurde hierfür eine Rechnerkopplung (A 5120 und A 7150) über IFFS-Leitung, wobei Kopplungen BC – BC, BC – AC oder AC – AC möglich sind. Obwohl das für diesen Kurs als Prinzipdemonstration entwickelte Programm in vieler Hinsicht perfektioniert werden kann, haben Versuche gezeigt, daß es auch in dieser Version mit praktischem Nutzen einsetzbar ist.

6.2.1. Ausgangspunkt

In Rechnernetzen ist es üblich, daß ein oder mehrere Rechner als sogenannte Server (etwa soviel wie Diener) fungieren. Ihre Aufgabe ist die Ausführung bestimmter Dienste, die über das Netz angefordert werden können (z. B. Recherchen in Datenbanken). Zu diesem Zweck muß ein Rechner über die Kommunikationsverbindung Aktionen auf einem anderen Rechner auslösen können. Das hier vorgestellte Programm zeigt, wie solche Dienste via Schnittstelle abgefordert werden können.

6.2.2. Das Konzept

Im Beispiel werden zwei Rechner bidirektional miteinander verbunden. Einer erhält das Rahmenprogramm für den Server und der andere das für den Klienten.

Zuerst muß auf dem Server der Loginprozeß

```
MAKE QUIT       ( QUIT ist ein DOER, der folgendes machen soll: )
>SYSTEM         ( Einschalten des Systemmodus                   )
CHECK           ( testen, ob alles in Ordnung ist; ggf. Ordnung )
                ( herstellen [z. B. Stapelüberlauf korrigieren] )
[COMPILE] [     ( Ausführmodus einschalten                      )
BEGIN           ( Beginn der Endlosschleife des Textinterpreters)
  RP0 RP!       ( Initialisierung des Rückkehrstapels           )
  PROMPT        ( Eingabebereitschaft signalisieren             )
  QUERY         ( Entgegennahme der Eingaben                    )
  >PROCESS      ( Nutzermodus aktivieren                        )
  INTERPRET     ( eigentliche Interpretation des empfangenen    )
                ( Textes; Ausführung/Kompilation und Fehler-    )
                ( test, ggf. Ausstieg zum Fehlersystem          )
  >SYSTEM       ( Umschaltung auf Systemmodus                   )
  ANSWER        ( Antwort, daß kein Fehler bemerkt wurde        )
AGAIN           ( unbedingt zurück zum Anfang der Schleife      )
;               ( dieses Semikolon wird deshalb nie erreicht    )
```

Bild 6.1 Kodierung von QUIT bei comFORTH 2

```
BLK    !         =0          !           <>0
       ! Entnahme des Eingabestroms erfolgt aus dem
       ! Texteingabepuffer !        Blockpuffer
-------+-------------------+-----------------------------
adr    !         TIB       ! Blockpufferadresse
#byte  !        #TIB       ! 1024=Blocklänge


---------------------------------------------------------
 !           !                          !
 !entnommen  !nächster   übernächster   !ungültig
 !           !                          !
---------------------------------------------------------
  ^          ^                           ^
  !          !                           !
 adr       adr+>IN                     adr+#byte
```

Bild 6.2 Herkunft des Eingabestroms (oben) und Zählweise der entnommenen Zeichen (unten)

(**LOGIN** auf der Datei SERVER.CF2) gestartet werden. Das gibt dem Klienten die Möglichkeit zur Aufnahme der Verbindung (ebenfalls mit LOGIN; Datei CLIENT.CF2). In unserem Beispiel erwartet der Server als erste Mitteilung ein Byte, das als Kennungsnummer des jeweiligen Klienten angesehen wird. Nach dessen Eintreffen wird zuerst die Codefeldadresse des Wortes **SFIND** und dann ein Quittungsbyte „Null" an den Klienten übermittelt; anschließend wird der Serverbetrieb durch **SERVE** aufgenommen. Von nun an kann der Klient den Server zu beliebigen Aktionen (Ausführung von Forthworten) anregen. Dafür findet die folgende Methode Verwendung: Der Klient wartet aktiv auf die Quittung „Null" vom Server und beginnt dann mit dem Senden von 16 Bit. Der Server befindet sich im Wort SERVE, wo die entgegengenommene Zahl als eine Codefeldadresse angesehen und das entsprechende Wort per **EXECUTE** ausgeführt wird. Anschließend wird wieder eine Null als Quittung an den Klienten zurückgesendet, die diesem die Bereitschaft für neue Serverdienste signalisiert.
Der Klient ist in der Lage, noch unbekannte Codefeldadressen in Erfahrung zu bringen, er kann so lernen, Dienst für Dienst rationeller als über den Namen in Anspruch zu nehmen. Dazu löst er zuerst die Ausführung des Serverwortes SFIND (siehe Datei CLIENT.CF2) aus. Danach wird mit **PUTS** der Name des zu suchenden Wortes als Zeichenkette übermittelt, die der Server mit **GETS** abholt und mit FIND in seinem Wörterbuch sucht. Als Ergebnis der Suche übergibt der Server zwei Zahlen mittels **PUT PUT** an den Klienten, der sie mit **GET GET** entgegennimmt. Ob das gesuchte Wort im Server gefunden werden konnte, zeigt im Klienten die Zahl auf dem Stapel als Flag an. Ist es „wahr", liegt auf dem SOS (second of stack) die dazugehörige Codefeldadresse, andernfalls ein uninteressanter Wert.
Damit der Bediener mit dem hier erläuterten Mechanismus nicht belastet wird, steht ihm das Definitionswort **SERVICE:** für die bequeme Inanspruchnahme von dem Namen nach bekannten Serverdiensten zur Verfügung. Es wird wir folgt benutzt:

SERVICE: Dienstname Serverwort

Nach der Definition von „Dienstname" kann durch dessen Aufruf die Ausführung des dazugehörigen „Serverwortes" veranlaßt werden. Bei der Definition von „Dienstname" wird im Klienten ein neuer Wörterbucheintrag eingerichtet. In dessen Parameterfeld befinden sich 2 Byte für ein Feld zur Aufnahme in die Liste aller Servicedienste (Ankerfeld **SERVICE^**), 2 Byte (mit Null initialisiert) zum Abspeichern der vom Server übermittelten Codefeldadresse und einige weitere Bytes für den Namen des korrespondierenden Serverwortes als Zeichenkette. Bei Aufruf von „Dienstname" (**DOES>**-Teil von SERVICE:) wird zuerst nachgeschaut, ob sich in der Zelle für die Codefeldadresse bereits ein von Null verschiedener Wert befindet. In diesem Fall kann sofort die Ausführung des Dienstes veranlaßt werden; andernfalls wird die Codefeldadresse erst mit SFIND in Erfahrung gebracht.

Das Wort **LOGOFF** ermöglicht ein Abmelden des Klienten vom Server. Dazu werden auf dem Server der Loginprozeß neu gestartet und anschließend die Codefeldadressen aller Serverdienste im Klienten mit Null überschrieben. So ist ein Wechsel des Servers möglich, ohne daß Konflikte aufgrund verschiedener Codefeldadressen für die einzelnen Dienste auftreten.

6.2.3. Zum Programmaufbau

Die Software beider Kommunikationspartner besteht im wesentlichen aus drei aufeinander aufbauenden Schichten. In der untersten Schicht wird die physische Bedienung der Schnittstelle ermöglicht. Dazu gehören die Funktionen **TX** „Byte senden", **TX?** „Senderstatus", **RX** „Byte empfangen", **RX?** „Empfängerstatus" und **0!X** „Schnittstelle initialisieren". Sie sind auf den Dateien IFSS5120.CF2 für den Bürocomputer A 5120 und IFSS 7150.CF2 für den Arbeitsplatzcomputer A 7150 codiert. Eine besondere Aufgabe kommt der physischen Initialisierung zu. Durch sie müssen (noch vor dem logischen LOGIN) nicht nur die Schnittstellen auf den gewünschten Sende- und Empfangsmodus eingestellt, sondern auch das „Einrasten" vollzogen werden. Das ist notwendig, da nach der Initialisierung der Peripheriebausteine die Sendepuffer nicht in jedem Fall leer sind.
In der zweiten Schicht (Datei TRANSFER.CF2) wird das überwachte Senden und Empfangen von Bytes (**CPUT** und **CGET**), von Worten (PUT und GET) und von Zeichenketten (PUTS und GETS) durchgeführt. Nur in der obersten Schicht unterscheiden sich die Programme von Server (SERVER!CF2) und Klient (CLIENT.CF2).

6.2.4. Ein Sitzungsbeispiel

Zuerst ist auf beiden Rechnern die Schnittstelle physisch mit **ALSO COM 0!X** zu initialisieren. Danach muß auf dem Server durch Aufruf des Wortes LOGIN der Loginprozeß gestartet werden. Damit ist der Server für den Klienten verfügbar. Von nun an arbeitet der Server ferngesteuert, und vom Klienten aus kann die Verbindung mit

12 LOGIN*(cr) ok*

aufgebaut werden, wobei die 12 die Kennung des Klienten darstellen soll.

Im folgenden Benutzungsbeispiel ist es das Ziel, einen Block vom Massenspeicher des Servers zu lesen und zum Klienten zu übertragen. Das erfordert erst noch die Vereinbarung eines Dienstes für die Aktion „Block lesen":

SERVICE: SBLOCK BLOCK*(cr) ok*

Nach der Übermittlung einer Blocknummer (hier 33) kann der Klient den Dienst **SBLOCK** veranlassen:

33 >S SBLOCK*(cr) ok*

Zur Übertragung des Speicherblocks zum Klienten sind noch die Parameter für **SPUTS** (PUTS auf dem Server) bereitzustellen. Die Anfangsadresse des zu übertragenden Datenbereichs von BLOCK liegt dort noch auf dem Stapel. Die Anzahl der zu sendenden Bytes entspricht der Länge eines Massenspeicherblocks (gleich 1024) und muß mit **>S** zum Stapel des Servers transferiert werden, bevor die Übertragung beginnen kann:

1024 >S SPUTS*(cr) ok*

Mit GETS nimmt nun der Klient die Daten in Empfang:

PAD 1024 GETS DROP*(cr) ok*

In diesem Beispiel wurden die Daten nach **PAD** gebracht. Die ganze Aktion ließe sich natürlich auch als Programm in einem Forthwort definieren:

```
: BLOCK> (ps: n ===>) (Block empfangen)
  >S SBLOCK 1024 >S SPUTS
  PAD 1024 GETS DROP ;
```

Der Klient kann nun auf bequeme Weise beliebige Blöcke vom Server holen:

33 BLOCK>*(cr) ok*

Schluß

Literatur

/1/ Brodie, L.: In Forth denken. München: Carl Hanser Verlag 1986

/2/ Vack, G.-U.: Der Standard FORTH-83. Suhl: Kammer der Technik 1987

✉ KONTAKT ☎

Wilhelm-Pieck-Universität Rostock, Sektion Technische Elektronik, Albert-Einstein-Straße 2, Rostock, 2500; Tel. 40 52 14

Tafel 6 Kurzbeschreibung der Forthworte

Name	Stapeleffekt	Beschreibung
COLD	(===>)	Kaltstart, Initialisierung des Forthsystems
QUIT	(===>)	Ausführungsmodus, Eingabebereitschaft, Wortinterpretation beginnen
#TIB	(===> addr)	Variable, enthält Anzahl der Bytes im Texteingabepuffer
TIB	(===> addr)	Adresse des Texteingabepuffers, wo Zeichen vom Eingabestrom gepuffert sind
BLK	(===>)	Variable, enthält Nummer des Massenspeicherblocks oder 0 für Tastatur
>IN	(===>)	Variable, enthält aktuellen Zeichenoffset im Eingabestrom
WORD	(c===> addr)	anhand Trennzeichen c Teilzeichenkette aus Eingabe isolieren und nach addr schaffen
ABORT	(===>)	Parameterstapel säubern und Ausführen von QUIT
ABORT"	(?===>)	Bei Ausführung vcn ABORT" text" wird bei ?=wahr text ausgegeben, dann ABORT.

```
   Datei A:CLIENT.LDR    Block 0
 0 CLIENT                                          Stand: EW 25-Oct-89
 1 ==================================================================
 2                        Lader für den Klienten
 3 ==================================================================
 4
 5
 6
 7
 8      CLIENT
 9
10
11
12
13
14
15
```

```
   Datei A:CLIENT.CF2    Block 1
 0 \ Rahmenprotokoll Klient                            RN 30-Oct-89
 1
 2  ONLY COM ALSO FORTH DEFINITIONS  DECIMAL
 3
 4 CR .( Warte ...lade Rahmenprotokoll Klient )
 5
 6    2 5 THRU
 7
 8 CR .(          ...fertig )
 9 CR
10
11  ONLY FORTH DEFINITIONS
12
13
14
15
```

```
   Datei A:CLIENT.CF2    Block 2
 0 \     'SFIND LOGIN SEXECUTE SFIND                   EW 25-Oct-89
 1
 2 VARIABLE 'SFIND  ( cfa<SFIND> des Servers ) 'SFIND OFF
 3
 4 : LOGIN      ( ps: 8b ===> )( Klient 8b einloggen )
 5              ( Protokoll: t:8b r:addr )
 6              CPUT GET 'SFIND ! ;
 7
 8 : SEXECUTE ( ps: addr ===> )( auf Server ausfuehren )
 9              ( Protokoll: r:8b t:addr )( letzter Fehler! )
10              CGET ?DUP IF ." Serverfehler " . QUIT THEN PUT ;
11
12 : SFIND      ( ps: addr ===> addr ? )( im Server suchen )
13              ( Protokoll: ..sexecute.. t:s r:addr r:? )
14              'SFIND @ ?DUP 0= ABORT" Server nicht geloggt"
15              SEXECUTE COUNT PUTS GET GET ;
```

```
   Datei A:CLIENT.CF2    Block 3
 0 \     SERVICE^ SERVICE:                             RN 30-Oct-89
 1
 2 VARIABLE SERVICE^ ( Anker Serviceliste )  SERVICE^ OFF
 3
 4 : SERVICE: ( ps: ===> )( ib: local-name remote-name )
 5            ( definiert Serverdienst local-name, im )
 6            ( Server wird remote-name ausgefuehrt )
 7            CREATE HERE  SERVICE^ DUP @ ,  !   0 ,
 8                   BL WORD C@ 1+ ALLOT DOES>
 9            ( ps: ===> )( Aufruf von remote-name in Server )
10            DUP 2+ @ ?DUP            \ 'cf schon bekannt?
11            IF   SWAP DROP
12            ELSE DUP 4 + SFIND 0=
13                 ABORT" unbekannter Dienst" DUP ROT 2+ !
14            THEN SEXECUTE ;
15
```

```
   Datei A:CLIENT.CF2    Block 4
 0 \     Standard-Dienstleistungen                     RN 30-Oct-89
 1
 2 SERVICE: SCPUT  CPUT            SERVICE: SCGET  CGET
 3 SERVICE:  SPUT   PUT            SERVICE:  SGET   GET
 4 SERVICE:  SPUTS  PUTS           SERVICE:  SGETS  GETS
 5 SERVICE: SLOGIN LOGIN
 6
 7 : >S ( ps: 16b ===> )( Transfer zum Server )
 8      ( Protokoll: ..sexecute.. t:16b )
 9      SGET  \ Abholen von 16 Bit durch den Server veranlassen
10      PUT ; \ 16 Bit (vom Klientenstapel zum Serverstapel) senden
11
12 : S> ( ps: ===> 16b )( Transfer vom Server )
13      ( Protokoll: ..sexecute.. r:16b )
14      SPUT  \ Senden von 16 Bit durch den Server veranlassen
15      GET ; \ 16 Bit (vom Serverstapel zum Klientenstapel) empf.
```

```
   Datei A:CLIENT.CF2    Block 5
 0 \     LOGOFF                                        RN 30-Oct-89
 1
 2 : LOGOFF ( ps: ===> )( Abmelden beim Server )
 3          ( Protokoll: ..sexecute.. ..login.. )
 4          SLOGIN  0 'SFIND !           \ Server-Restart
 5          SERVICE^ BEGIN  @ ?DUP       \ Listenende?
 6                   WHILE  0 OVER 2+ ! \ 'cf-Feld ruecksetzen
 7                   REPEAT ;
 8
 9
10
11
12
13
14
15
```

```
   Datei A:SERVER.LDR    Block 1
 0 \ Ladeblock
                                                       EW 25-Oct-89
 1
 2  ONLY FORTH DEFINITIONS  DECIMAL
 3
 4 CR CR  .( -------- Lader fuer den Server )
 5
 6         @:ASM86      1 INCLUDE   \ 8086-Assembler
 7         @:IFSS7150   1 INCLUDE   \ Schnittstellenbedienung
 8         @:TRANSFER   1 INCLUDE   \ Transferdienste
 9         @:SERVER     1 INCLUDE   \ Serverinterface
10
11 CR      .( -------- fertig )
12 CR
13
14
15
```

```
   Datei A:IFSS7150.CF2  Block 1
 0 \ Schnittstellenbedienung fuer A 7150               RN 30-Oct-89
 1
 2  ONLY FORTH DEFINITIONS  DECIMAL
 3
 4 CR .( Warte ...lade physische Schnittstellenbedienung )
 5
 6    2 5 THRU
 7
 8 CR .(          ...fertig )
 9 CR
10
11
12
13
14
15
```

```
   Datei A:IFSS7150.CF2  Block 2
 0 \     COM AWAIT O!X                                 EW 25-Oct-89
 1
 2 ALSO ROOT DEFINITIONS   VOCABULARY COM
 3      COM  DEFINITIONS
 4
 5 CODE COM ( ps: ax dx ===> ax )( COM-Interrupt ansprechen )
 6          DX AX MOV, BX DX MOV,
 7          20 # INT,  \ Intercupt 14H - COM-Schnittstelle BIOS
 8          AX BX MOV, DX POP, NEXT,
 9          END-CODE
10
11 : IFSS     ( ps: ax ===> ax )( Kanal COM2 )
12            1 COM ;
13
14
15
```

```
   Datei A:IFSS7150.CF2  Block 3
 0 \     SUBFUNCTION (O!IFSS) (>IFSS) (IFSS>) IFSS? ...EW 25-Oct-89
 1
 2 : SUBFUNCTION ( ps: 8b ===> )( ib: name )( Unterfunktion def.)
 3               CREATE C, DOES>
 4               ( ps: al ===> ax )( Funktion 8b an IFSS )
 5               C@ JOIN IFSS ;
 6
 7 0 SUBFUNCTION (O!IFSS)
 8 1 SUBFUNCTION (>IFSS)
 9 2 SUBFUNCTION (IFSS>)
10 3 SUBFUNCTION IFSS?
11
12 : RX     ( ps: ===> 8b )( Zeichen lesen )
13          0 (IFSS>) 255 AND ;
14 : TX     ( ps: 8b ===> )( Zeichen schreiben )
15          (>IFSS) DROP ;
```

```
   Datei A:IFSS7150.CF2  Block 4
 0 \     CTRL %RX? %TX? O!IFSS RX? TX?                 HP 24-Oct-89
 1
 2 2 BASE !
 3 11101011 CONSTANT CTRL ( Steuerwort: 8 Bit Daten, 1 Stop )
 4                        ( mit ungeradem Paritaetsbit )
 5 0000000100000000 CONSTANT %RX? ( Empfangsdaten )
 6 0010000000000000 CONSTANT %TX? ( Sender leer )
 7
 8 DECIMAL
 9 : O!IFSS ( ps: ===> )( Initialisierung )
10          CTRL (O!IFSS) DROP ;
11 : RX? ( ps: ===> ? )( ungleich Null bei Empfangsdaten vorh. )
12       0 IFSS? %RX? AND ;
13 : TX? ( ps: ===> ? )( ungleich Null bei Sender leer )
14       0 IFSS? %TX? AND ;
15
```

```
   Datei A:IFSS7150.CF2  Block 5
 0 \ Initialisieren der Serverschnittstelle            HP 24-Oct-89
 1
 2 HEX
 3
 4 : O!X    ( ps: ===> )( Initialisierung der Schnittstelle )
 5          ( zusammen mit Einrasten )
 6          O!IFSS 5 0 DO 0 TX LOOP 55 TX ;
 7
 8 DECIMAL
 9
10
11
12
13
14
15
```

```
Datei A:SERVER.LDR     Block 0
 0 SERVER                                              Stand: EW 25-Oct-89
 1 ==================================================================
 2                     Lader für den Server
 3 ==================================================================
 4
 5
 6
 7
 8        SERVER
 9
10
11
12
13
14
15
```

```
Datei A:CLIENT.LDR     Block 1
 0 \ Ladeblock                                                  EW 25-Oct-89
 1
 2  ONLY FORTH DEFINITIONS  DECIMAL
 3
 4 CR CR   .( -------- Lader fuer den Klienten )
 5
 6        @:IFSS5120   1 INCLUDE    \ Schnittstellenbedienung
 7        @:TRANSFER   1 INCLUDE    \ Transferdienste
 8        @:CLIENT     1 INCLUDE    \ Rahmenprotokoll Klient
 9
10 CR      .( -------- fertig )
11 CR
12
13
14
15
```

```
Datei A:IFSS5120.CF2  Block 1
 0 \ Schnittstellenbedienung fuer A 5120                    RN 30-Oct-89
 1
 2  ONLY FORTH DEFINITIONS  HEX
 3
 4 CR .( Warte ...lade physische Schnittstellenbedienung )
 5
 6    2 4 THRU
 7
 8 CR .(           ...fertig )
 9 CR
10
11  ONLY FORTH DEFINITIONS  DECIMAL
12 \S ------------------------------------------------------------------
13 SIO auf der ASS des BC 5120, DFUe-Kanal: Tornummern 52H und 53H
14 Am Koppelbus ist eine Wickelverbindung vom CTC der ZRE zur SIO
15 der ASS erforderlich.
```

```
Datei A:IFSS5120.CF2  Block 2
 0 \ COM Tornummern & Initialisieren PIO, CTC              RN 30-Oct-89
 1 \ kleines Glossar fuer die Bedienung der Tore:
 2 \ CP!  ( ps: 8b port ===> )  Torausgabe von 8b auf "port"
 3 \ CP@  ( ps: port ===> 8b )  8b lesen vom Tor "port"
 4
 5 ALSO ROOT DEFINITIONS   VOCABULARY COM
 6      COM  DEFINITIONS
 7
 8 52 CONSTANT SIO    ( Datentor der SIO )
 9 53 CONSTANT PSIO   ( Programmier- & Statustor der SIO )
10 OC CONSTANT CTC    ( Tor Kanal 0 des CTC auf ZRE K 2526 )
11
12 : SIOINIT     ( ps: ===> )
13   OEA 05 OC1 3 0 1 45 4 18 ( Reset)  9 0 DO PSIO CP! LOOP ;
14
15 : CTCINIT     ( ps: ===> )                17 CTC CP!   1 CTC CP! ;
```

```
Datei A:IFSS5120.CF2  Block 3
 0 \ RX? TX? RX TX                                          HP 24-Oct-89
 1
 2 : RX?    ( -- ?)( wahr, falls Zeichen eingegangen & abholbereit)
 3   PSIO CP@ 1 AND ;
 4
 5 : TX?    ( -- ?)( wahr, falls Sender frei & senden moeglich)
 6   PSIO CP@ 4 AND ;
 7
 8 : RX     ( -- c)( eingegangenes Zeichen aus Empfangspuffer lesen)
 9   SIO CP@ ;
10
11 : TX     ( c --)( Zeichen zum Senden in Sendepuffer schreiben)
12   SIO CP! ;
13
14
15
```

```
Datei A:IFSS5120.CF2  Block 4
 0 \ Einrasten der Schnittstelle                            RN 30-Oct-89
 1
 2 : O!X    ( ps: ===> )( Initialisierung der Schnittstelle )
 3          ( zusammen mit Einrasten )
 4   CTCINIT SIOINIT
 5   BEGIN BEGIN RX? UNTIL RX    O= UNTIL
 6   BEGIN BEGIN RX? UNTIL RX 55 = UNTIL ;
 7
 8
 9
10
11
12
13
14
15
```

```
Datei A:SERVER.CF2     Block 1
 0 \ Rahmenprotokoll Serverinterface                        RN 30-Oct-89
 1
 2  ONLY COM ALSO FORTH DEFINITIONS  DECIMAL
 3
 4 CR .( Warte ...lade Rahmenprotokoll Server )
 5
 6    2 3 THRU
 7
 8 CR .(           ...fertig )
 9 CR
10
11  ONLY FORTH DEFINITIONS
12
13
14
15
```

```
Datei A:SERVER.CF2     Block 2
 0 \     ERROR? SERVE                                       RN 30-Oct-89
 1
 2 : ERROR?  ( ps: ===> 8b )( ermittelt Fehlernummern )
 3   0 ;      ( erster Ansatz: Eigentest immer in Ordnung )
 4
 5 : SERVE  ( ps: ===> ... )( Dienstleistungen ueber Interface )
 6          ( Protokoll: r:addr ... t:8b )
 7          RPO RP!               \ Rueckkehrstapel initialisieren
 8          BEGIN  GET EXECUTE    \ Dienstleistung annehmen
 9                 ERROR? CPUT    \ Eigentest und Quittung senden
10          AGAIN  ;
11
12
13
14
15
```

```
Datei A:SERVER.CF2     Block 3
 0 \     SFIND CLIENT# LOGIN                                RN 30-Oct-89
 1
 2 : SFIND   ( ps: ===> )( 'cf-Suche ueber Interface )
 3           ( Protokoll: r:s t:addr t:? )
 4           PAD 1+ DUP 32 BLANK         \ PAD mit Leerzeichen fuellen
 5           31 GETS PAD C!              \ Wortname zum PAD empfangen
 6           PAD FIND SWAP PUT PUT ; \ Wort suchen; Ergebnis senden
 7
 8 VARIABLE CLIENT#
 9 : LOGIN   ( ps: ===> )( Verbindung mit Klienten aufbauen )
10           ( Protokoll: r:8b t:addr t:8b ..serve.. )
11           CGET CLIENT# C!     \ Nr. des Klienten holen & merken
12           ['] SFIND PUT       \ 'cf von SFIND senden
13           0 CPUT              \ Blindquittung senden
14           SERVE ;             \ Serverbetrieb aufnehmen
15
```

```
Datei A:TRANSFER.CF2  Block 1
 0 \ Transferdienste                                        RN 30-Oct-89
 1
 2  ONLY FORTH ALSO COM DEFINITIONS  DECIMAL
 3
 4 CR .( Warte ...lade Transferfunktionen )
 5
 6    2 3 THRU
 7
 8 CR .(           ...fertig )
 9 CR
10
11  ONLY FORTH DEFINITIONS
12
13
14
15
```

```
Datei A:TRANSFER.CF2  Block 2
 0 \      (CPUT) (CGET) CPUT CGET PUT GET                   RN 30-Oct-89
 1 \ SPLIT   ( ps: 16b ===> 8b-low 8b-high )  16b in 2x 8b zerlegen
 2 \ JOIN    ( ps: 8b-low 8b-high ===> 16b )  2x 8b zusammenfuegen
 3
 4 : (CPUT) ( ps: 8b ===> )( synchronisiertes Senden )
 5          BEGIN  TX?  UNTIL  TX ;
 6 : (CGET) ( ps: ===> 8b )( synchronisierter Empfang )
 7          BEGIN  RX?  UNTIL  RX ;
 8 : CPUT   ( ps: 8b ===> )( ueberwachtes Senden )
 9          DUP (CPUT) (CGET) - ABORT" Uebertragungsfehler" ;
10 : CGET   ( ps: ===> 8b )( ueberwachter Empfang )
11          (CGET) DUP (CPUT) ;
12 :  PUT   ( ps: 16b ===> )( Wort senden )
13          SPLIT SWAP CPUT CPUT ;
14 :  GET   ( ps: ===> 16b )( Wort empfangen )
15          CGET CGET JOIN ;
```

```
Datei A:TRANSFER.CF2  Block 3
 0 \      PUTS GETS                                         RN 31-Oct-89
 1
 2 : PUTS ( ps: addr u ===> )( Datenblock senden )
 3        DUP PUT OVER + SWAP
 4        ?DO   I C@ CPUT   LOOP ;
 5
 6 : GETS ( ps: addr u1 ===> u2 )( Datenblock empfangen )
 7        ( maximale Pufferlaenge u1, Laenge des Blocks u2 )
 8        GET DUP >R UMIN R@ OVER - -ROT OVER + SWAP
 9          ?DO   CGET I C!   LOOP
10        0 ?DO   CGET DROP   LOOP   R> ;
11
12
13
14
15
```